FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Orquestração e aprovisionamento de um estúdio televisivo baseado na tecnologia IP na Cloud

Vasco Fernandes Gonçalves

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Rui Filipe Lima Maranhão de Abreu

21 de Julho de 2017

Orquestração e aprovisionamento de um estúdio televisivo baseado na tecnologia IP na Cloud

Vasco Fernandes Gonçalves

Mestrado Integrado em Engenharia Informática e Computação

21 de Julho de 2017

Resumo

Com a evolução que se tem verificado nos últimos anos ao nível da velocidade de transmissão de dados em redes de computadores, tem surgido a possibilidade de transitar a produção de conteúdos televisivos, tradicionalmente produzidos em dispositivos específicos para o efeito e com elevados custos, para sistemas informáticos interligados com redes IP.

A acompanhar esta evolução tem-se também assistido a novas tendências na produção de conteúdo televisivo, como o aumento da resolução da imagem, transmissão de mais canais de som (*surround*, múltiplas línguas) ou a transmissão de novos formatos de conteúdo, como 3D, 360° ou realidade virtual.

O elevado custo do equipamento tradicional, principalmente no que diz respeito à produção de conteúdo televisivo ao vivo, pode dificultar a entrada no mercado de novos canais ou a adoção de novos tipos de media por canais já existentes.

A evolução ao nível da performance do hardware informático, velocidade das redes e algoritmos para compressão e transmissão de conteúdo multimédia veio possibilitar a virtualização destes ambientes, de forma permitir melhorar o aproveitamento dos recursos disponíveis e até a alocá-los dinamicamente de acordo com a variação da sua necessidade.

Assim, este trabalho propõe minimizar os custos previamente referidos através da criação de uma plataforma que permita lançar numa *cloud* um sistema virtualizado que desempenhe as mesmas funções que os equipamentos tradicionais. Esta deve providenciar métricas de funcionamento do mesmo e funcionar de forma intuitiva para o utilizador.

Abstract

With the evolution of data transmission speeds on computer networks in recent years, the possibility to move the production of television content, traditionally produced in specific devices for this purpose and at high costs, to IP network connected systems has appeared.

In addition, new trends in the production of television content such as increased image resolution, transmission of more sound channels (surround, multiple languages) or the transmission of new content formats such as 3D, 360° or virtual reality.

The high cost of traditional equipment, particularly with regard to the production of live television content, may hinder the entry of new channels into the market or the adoption of new types of media by existing channels.

The evolution in the performance of computer hardware, network speed and algorithms for compression and transmission of multimedia content has enabled the virtualisation of these environments, so as to improve the utilization of available resources and even to allocate them dynamically according to the variation of their need.

Thus, this paper proposes to minimize the costs previously mentioned by creating a platform that enables the deployment in a cloud of a virtualized system that performs the same functions as the traditional equipment. This should provide metrics of its operation and be intuitive for the user.

Thus, this work proposes to minimize the costs previously mentioned by creating a platform that enables the deployment in a cloud, in a manner that is intuitive for a user, of a virtualized system that performs the same functions as the traditional equipment, and also provide metrics of its performance.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, à minha família e amigos pelo apoio incondicional ao longo de todo o meu percurso académico.

A todos os elementos e direção da MOG que me acompanharam e apoiaram nesta etapa da minha formação. Ao Miguel Poeira, Pedro Ferreira e Pedro Santos, pela importante supervisão e orientação, sem a qual teria sido impossível a elaboração deste projeto.

Ao Professor Rui Maranhão pelo contributo a nível técnico e académico, e pela paciente revisão dos conteúdos desta dissertação.

À instituição Faculdade de Engenharia pelo conhecimento que me foi transmitido, por tudo aquilo que representa, pelos valores que me incutiu e pela excelência nas condições disponibilizadas durante o meu ciclo de estudos.

Finalmente aos colegas André Regado, António Presa, Bruno Pereira, Pedro Rocha e Vasco Filipe, pelo clima de inter-ajuda e camaradagem que proporcionaram no quotidiano deste último semestre.

Vasco Gonçalves

"One original thought is worth a thousand mindless quotings."

Diogenes Laërtius

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

API	Application Programming Interface
AWS	Amazon Web Services
CIDR	Classless Inter-Domain Routing
CLI	Command Line Interface
CNI	Container Network Interface
CNM	Container Network Model
CPU	Central Processing Unit
DASH	Dynamic Adaptative Streaming over HTTP
DNS	Domain Name Service
FPS	Frames per second (Fotogramas por segundo)
HD	High Definition
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IEEE	Institute of Electrical and Electronics Engineers
IGMP	Internet Group Management Protocol
IP	Internet Protocol
IT	Information Technology
JSON	JavaScript Object Notation
JT-NM	Joint Task Force on Networked Media
LACP	Link Aggregation Control Protocol
LAN	Local Area Network
LXC	Linux Containers
MPEG	Moving Picture Experts Group
MPEG-DASH	Dynamic Adaptive Streaming over HTTP
MPEG-TS	MPEG Transport Stream
NAT	Network Address Translation
NIC	Network Interface Controller
NIST	National Institute of Standards and Technology
NMOS	Networked Media Open Specification
OB	Outside Broadcasting
OS	Operating System (Sistema Operativo)
REST	Representational State Transfer
RFC	Request for Comments
RTP	Real-time Transport Protocol

SDI	Serial Digital Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
VM	Virtual Machine
VXLAN	Virtual eXtensible LAN
XML	eXtensible Markup Language

Capítulo 1

Introdução

Na produção de conteúdos televisivos existem duas grandes áreas, sendo elas as emissões ao vivo e a transmissão de conteúdo pré-produzido. Em produções ao vivo, sobre as quais nos focamos mais nesta dissertação, existem alguns requisitos fundamentais como a entrega da informação em tempo útil, com o menor atraso possível e com uma fidelidade visual e auditiva que garanta a qualidade da experiência do telespectador.

Atualmente a produção de conteúdo ao vivo acarreta custos elevados no que diz respeito ao equipamento necessário [**?**]. Tipicamente, este conteúdo é produzido *in loco* sendo o sinal posteriormente transmitido para o estúdio televisivo. Como exemplo, na transmissão de um evento desportivo existem múltiplas câmaras que estão ligadas a um carro de exteriores, ou *Outside Broadcasting (OB) van*, onde é feita a produção e a transmissão do programa para o estúdio. Estas *OB Vans* custam entre vários milhares até milhões de euros [**?**].

O surgimento de novos protocolos que melhoram a transmissão de conteúdo televisivo através de redes Ethernet bem como de equipamento de rede mais rápido potencia a utilização de ambientes informáticos descentralizados na *cloud* para a criação de estúdios televisivos virtuais [**?**].

A transição dos estúdios tradicionais para aplicações deste tipo pode trazer várias vantagens, quer a nível de custos quer a nível logístico. Dado que os eventos a cobrir são muitas vezes esporádicos, a maior agilidade que advém da utilização do *cloud computing* pode trazer outra flexibilidade aos canais televisivos no que diz respeito à cobertura de eventos.

Este projeto utiliza como principal cenário de teste o trabalho realizado por Miguel Poeira [**?**] que, recorrendo à tecnologia de *containers Docker*, implementa alguns módulos de um estúdio televisivo. Estes módulos distribuem entre si sinais de vídeo não comprimidos através de transmissões *multicast*. Se por um lado isto trás vantagens a nível da qualidade de imagem e facilidade de pós produção [**?**], o elevado volume de informação e a eventual distância entre as diferentes máquinas físicas na rede cria dificuldades nestas transmissões.

Para além disso, a utilização destes ambientes pode ter uma curva de aprendizagem elevada, principalmente para profissionais com conhecimentos limitados a nível informático, de áreas como a produção televisiva. É necessário coordenar uma infraestrutura descentralizada, com várias máquinas, e o *deployment* de uma aplicação pode não ser tão intuitivo como o desejado.

Assim, este projeto pretende propor a criação de uma plataforma que simplifique o processo de instanciação, manutenção e monitorização de um estúdio televisivo na *cloud* por parte de um realizador ou outro profissional da área da televisão. Importa apontar que apesar do caso de teste ser o dos estúdios de televisão virtuais, foi feito um esforço para tornar a plataforma genérica o suficiente para ser adaptável a outros cenários.

Neste documento é discutida a orquestração e aprovisionamento destes estúdios televisivos virtuais numa *cloud* privada, bem como as considerações tomadas a nível da configuração da mesma. Em paralelo com esta dissertação foi desenvolvida, pelo colega António Presa [**?**], uma interface que pretende disponibilizar os recursos desta infraestrutura de forma intuitiva.

1.1 Enquadramento

Esta dissertação foi realizada na empresa MOG Technologies, situada na Maia, que está no mercado há 15 anos, e tem como principal missão desde o primeiro dia a redução no custo da produção de conteúdos televisivos. Esta empresa, que tipicamente está mais envolvida no desenvolvimento de soluções para ambientes de pós-produção, tem vindo a apostar em novas tecnologias nomeadamente na área da produção de conteúdo ao vivo sobre IP.

O trabalho desenvolvido nesta dissertação insere-se num projeto europeu atualmente em desenvolvimento nesta empresa, o *Cloud Setup*, envolvendo ainda o INESC TEC e o HASLab.

Apesar de se basear num protótipo, este trabalho dá continuidade ao esforço da MOG em acompanhar as tendências do mercado na sua área de negócio. Para além disto, devido à possibilidade de generalização desta solução, esta poderá ter aplicabilidade na empresa ao nível dos processos internos de desenvolvimento e testes.

1.2 Motivação e Objetivos

A motivação para esta dissertação parte do já referido potencial para a redução dos custos e da carga logística para os canais de televisão na emissão de programas em direto, recorrendo a estúdios televisivos em ambientes virtualizados na *cloud*. Conforme foi anteriormente referido, há espaço para tornar a utilização destes recursos mais intuitiva para utilizadores com menor nível de conhecimentos a nível informático.

Assim, as principais contribuições deste projeto passam por:

- Investigação do estado de arte sobre as tecnologias que possibilitam a criação e manutenção de uma *cloud* privada
- Identificação das necessidades específicas para o *deployment* de um estúdio televisivo virtual

- Criação de um formato que descreva e especifique um estúdio televisivo virtual e permita a comunicação entre a interface e a infraestrutura
- Configuração de uma *cloud* privada que corra os módulos já existentes
- Implementação e disponibilização dos serviços necessários para a comunicação com a interface
- Validação do desenvolvimento através de testes de performance e do *deployment* de vários cenários, avaliando o correto funcionamento dos diferentes módulos

1.3 Estrutura do documento

Para além desta introdução, este documento irá conter mais cinco capítulos:

O capítulo 2, sobre o estado da arte investigado e o trabalho relacionado.

O capítulo 3, que aborda o cenário de teste considerado, nomeadamente a sua arquitetura e funcionalidades.

O capítulo 4, onde é descrito o problema e a solução implementada, incluindo a sua arquitetura e as tecnologias utilizadas.

O capítulo 5, onde são feitos testes de validação da solução implementada.

Finalmente o capítulo 6, onde se apresentam as conclusões retiradas do desenvolvimento deste projeto e o trabalho futuro.

São também incluídos em anexo

Capítulo 2

Estado da arte

Neste capítulo vamos explorar o estado da arte em tecnologias e sistemas para orquestração e aprovisionamento de serviços informáticos, como virtualização, *cloud computing*, orquestradores e redes Ethernet. Serão revistos os principais entraves à transição para estúdios televisivos virtuais e que evoluções têm ocorrido neste sentido.

2.1 Computação na *cloud*

Computação na *cloud* define-se como um serviço que disponibiliza recursos computacionais *on-demand*, acessíveis pelos seus utilizadores a partir de qualquer parte do mundo [**?**].

De acordo com o NIST, as suas características essenciais são:

- **Serviço *self-service on-demand***: os recursos podem ser providenciados automaticamente ao utilizador, sem intervenção humana por parte do provedor.
- **Amplo acesso de rede**: os recursos são facilmente acessíveis na rede, em vários tipos de dispositivos.
- **Agrupamento de recursos**: os recursos do provedor estão configurados de forma a servir múltiplos utilizadores (*multi-tenancy*), e são alocados de forma dinâmica de acordo com a procura.
- **Rápida elasticidade**: os recursos podem escalar, em certos casos automaticamente, de acordo com a procura do utilizador, aparentando ser ilimitados.
- **Serviço monitorizado**: os sistemas automaticamente retiram métricas da utilização dos recursos, utilizando-as para optimizar os mesmos e reportando-as, dando assim maior transparência ao provedor e utilizador.

Tem-se assistido a uma transformação na forma de disponibilização e consequente consumo de software. Cada vez mais a *cloud* é utilizada para abstrair certas camadas de uma aplicação.

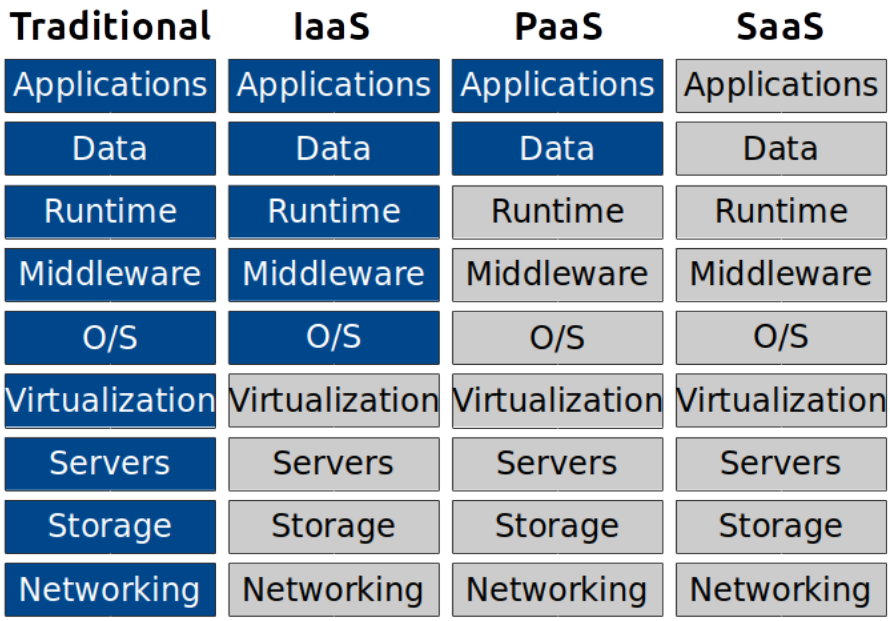

Figura 2.1: Modelos de *cloud computing*

Prevê-se até que a computação se possa tornar mais um serviço fundamental como eletricidade ou água (*utility*) [**?**].

Existem vários modelos de *cloud computing*, com níveis crescentes de abstração e virtualização (ver figura 2.1):

- **Tradicional**: toda a infraestrutura é gerida pelo utilizador.
- **Infrastructure as a Service (IaaS)**: os recursos computacionais básicos (como processamento, rede ou armazenamento) são providenciados, e o utilizador pode executar software arbitrário, incluindo sistemas operativos e aplicações.
- **Platform as a Service (PaaS)**: para além dos recursos básicos, os sistemas operativos também são providenciados. O utilizador continua a configurar as suas próprias aplicações, e pode ter algum controlo sobre as mesmas através de ficheiros de configuração.
- **Software as a Service (SaaS)**: todas as camadas são abstraídas. O utilizador não tem controlo das aplicações, acedendo às mesmas através de diferentes dispositivos clientes.

Estão ainda definidos quatro modelos de *deployment*, que dizem respeito aos utilizadores, entidade responsável e localização da *cloud*:

- **Cloud privada**: destina-se ao uso interno de uma organização, pode ser gerida pela própria ou outra entidade, e pode existir dentro ou fora das instalações da mesma.
- **Cloud comunitária**: destina-se ao uso por parte de uma comunidade com fins em comum e pode ser gerida e estar localizada nas instalações de uma ou mais das organizações intervenientes.

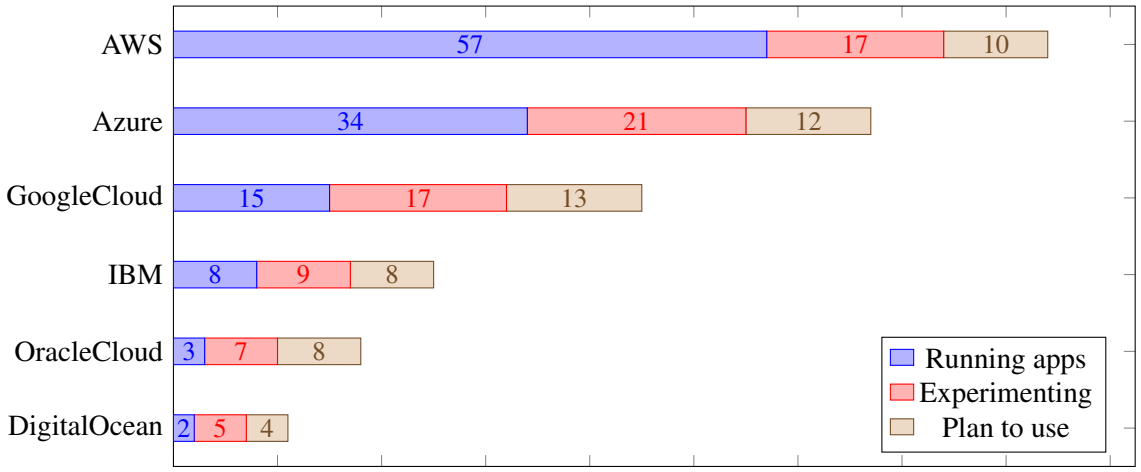

Figura 2.2: Utilização de provedores de *clouds* públicas (percentagem dos inquiridos) [**?**]

- **Cloud pública**: destina-se ao uso do público geral, pode ser gerida por uma entidade comercial, académica ou governamental e está localizada nas instalações dessa entidade.
- **Cloud híbrida**: é uma combinação de dois ou mais tipos de *clouds* anteriores, que apesar de existirem separadamente estão ligadas por tecnologias que permitem a portabilidade entre elas.

2.1.1 Provedores de *clouds* públicas

Com a popularização dos serviços de computação na *cloud* e a aproximação do anteriormente referido modelo de computação como uma *utility*, têm vindo a surgir diferentes provedores destes serviços. Estas *clouds* públicas permitem aos clientes subcontratar as suas necessidades de computação, e pagar apenas de acordo com a sua utilização, oferecendo grandes vantagens a nível de custos e fiabilidade [**?**].

Estes provedores oferecem soluções para o mesmo fim, mas muitas vezes utilizando diferentes abordagens a nível de infraestrutura, preço e modelo de utilização. Alguns dos mais utilizados são o AWS da Amazon, Microsoft Azure e Google Cloud (ver figura 2.2).

2.2 DevOps - Gestão de *clouds*

DevOps é a área que faz a ponte entre a área de desenvolvimento de software e a de IT. O seu nome deriva da junção das palavras *Development* e *Operations*. Tem como objetivo promover a melhoria dos processos a nível empresarial, entre as equipas de desenvolvimento e as de IT.

Os profissionais desta área são responsáveis por permitir o rápido teste, *deployment* e posterior monitorização de aplicações, que estão muitas vezes distribuídas em *clusters* de grande escala.

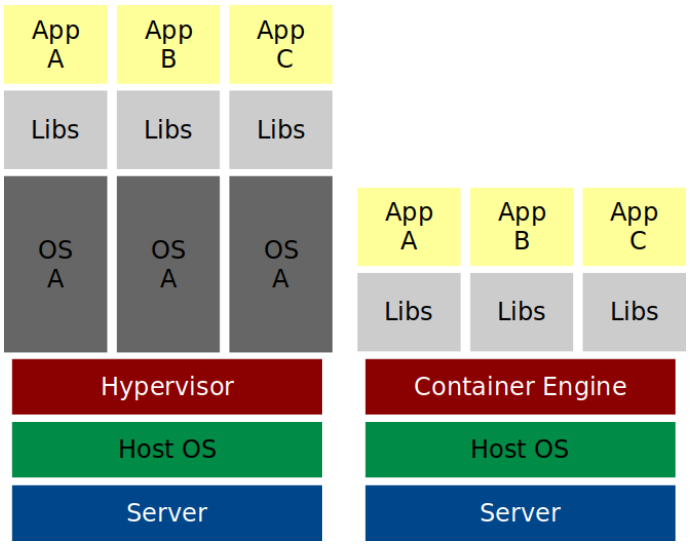

Figura 2.3: VMs vs *containers*

Assim, torna-se importante a utilização de ferramentas que permitam ter uma visão geral do estado da infraestrutura.

Estas ferramentas dividem-se em alguns grupos fundamentais:

- **Monitorização**: ferramentas que proporcionam uma visão global do consumo de recursos e do estado de funcionamento, tanto das máquinas físicas como do conteúdo virtualizado, recolhendo, armazenando e facilitando a visualização desta informação.

- **Alertas**: através destes dados de monitorização, é importante gerar alertas, que informam os administradores do sistema, de forma automática, assim que se manifeste algum problema. Estas ferramentas auxiliam neste processo, comunicando as falhas através de canais como e-mail ou serviços de mensagens instantâneas.

- **Logging**: a recolha de *logs* das aplicações é um importante mecanismo para a monitorização do funcionamento das mesmas. Devido à potencial escala dos *clusters*, utilizam-se ferramentas que recolhem os *logs* das diferentes máquinas ou *containers* e os armazenam, indexando o seu conteúdo para permitir a filtragem destes para posterior análise por parte dos administradores do sistema.

2.3 Virtualização

Existem duas tecnologias comummente utilizadas para disponibilizar a virtualização necessária à partilha de recursos em ambientes de *cloud computing*: Hipervisores e *Containers*. Ambas as tecnologias cumprem os requisitos de *multi-tenancy* e isolamento destas plataformas, no entanto existem algumas diferenças fundamentais na sua arquitetura (ver figura 2.3).

2.3.1 Hipervisores

Os Hipervisores são usados para o *deployment* de aplicações em máquinas virtuais (*virtual machines, VMs*). Nesta tecnologia, amplamente utilizada no *cloud computing*, cada uma das máquinas tem um sistema operativo instalado, o que permite maior flexibilidade no aprovisionamento de aplicações que necessitem de diferentes OSs ou de diferentes versões destes.

Apesar desta flexibilidade, uma máquina virtual tem certas desvantagens em relação a um *container*. São mais difíceis de administrar, devido ao trabalho acrescido da instalação de um sistema operativo e à fragmentação resultante do uso de diferentes versões dos mesmos. Têm geralmente pior desempenho e consomem muito mais recursos computacionais, devido ao *overhead* de correr um sistema operativo [**?**].

2.3.2 Containers

Ao contrário das máquinas virtuais, os *containers* não correm um sistema operativo completo, partilhando o da máquina "hospedeira" (máquina física ou virtual onde este está instanciado). Por um lado, o aprovisionamento de aplicações neste tipo de tecnologia é mais limitado, ao não permitir a execução de diferentes OSs. Ganha no entanto a nível de desempenho, consumindo muito menos recursos e tendo menos *overhead* computacional. É também mais ágil a nível de escalabilidade, tanto horizontal como vertical, ao ser mais simples a alocação dinâmica de recursos e ao ser consideravelmente mais rápido arrancar *containers* do que VMs. [**?**]

Uma grande vantagem da utilização de *containers* é a possibilidade de isolar cada módulo de uma aplicação, cada um com as suas configurações e dependências, evitando assim problemas de incompatibilidade entre diferentes componentes. Esta abordagem permite maior controlo sobre o ambiente de *deployment*, tornando este processo mais previsível.

A tecnologia de *containers* Linux (LXC) surge em 2008 com a introdução, na versão de *kernel* 2.6.24, da funcionalidade *cgroups*, que permite limitar os recursos disponíveis aos processos (como CPU, memória ou rede) [**?**]. Esta gestão de recursos, aliada à utilização de *namespaces* que isolam um grupo de processos dos restantes no sistema, permite a virtualização de aplicações sem ser necessário instanciar máquinas virtuais.

Nos últimos anos tem-se assistido a um grande crescimento no mercado da tecnologia de *containers*. Este mercado gerou, no ano de 2016, 762 milhões de dólares em receitas e é esperado que em 2020 chegue aos 2700 milhões [**?**] (ver figura 2.4).

2.3.2.1 Containers Windows

Na versão 2016 do sistema operativo Windows Server, foram introduzidos os *containers* Windows, com um conceito muito semelhante ao do Docker (ver secção 2.3.4). Um ou mais *hosts* podem instanciar *containers*, definidos com imagens, que estão contidas em repositórios. Para além da Powershell, também pode ser utilizada a interface linha de comandos do Docker para gerir esta plataforma, utilizando instruções iguais às da versão Windows.

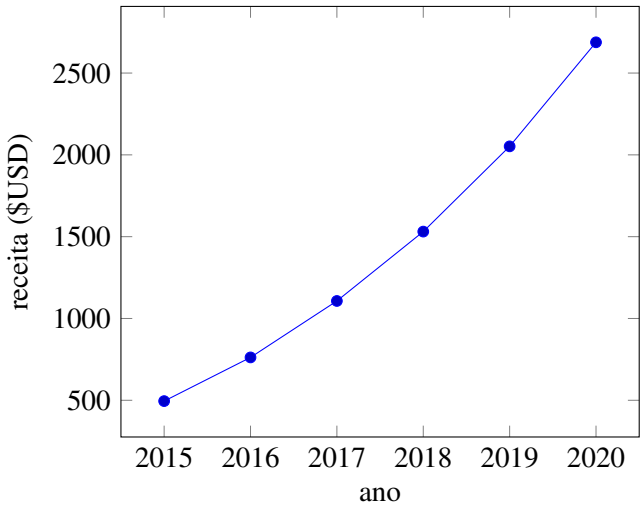

Figura 2.4: Evolução do mercado de *containers* [**?**]

Talvez reforçando a importância e ubiquidade dos *containers* na atualidade, a Microsoft fez uma parceria comercial com os criadores desta plataforma de forma a garantir o suporte dos seus sistemas operativos a esta tecnologia.

2.3.3 Host OS

Conforme já discutido, os *containers* partilham o *kernel* do sistema operativo hospedeiro. Assim, se na utilização de máquinas virtuais a escolha do sistema operativo pode ter em conta aspectos mais gerais como o desempenho e a estabilidade, na utilização de *containers* é necessário, não descurando estes aspectos, escolher um sistema operativo compatível com as imagens que queremos correr.

Para além disso, características como as versões do *kernel*, programas disponíveis, suporte da distribuição, entre outros, devem ser tidos pois podem influenciar o funcionamento e desempenho das soluções de virtualização.

2.3.4 Container engines

Apesar do LXC permitir o *deployment* de aplicações em *containers*, esta é uma tecnologia de baixo nível. Têm surgido alguns desenvolvimentos em soluções que o utilizam como base, estendendo as suas funcionalidades. Uma destas soluções é o Docker.

O Docker é uma das implementações mais populares de um *container engine*. Assenta numa arquitetura cliente-servidor, composta por um *daemon*, uma API REST e por uma aplicação cliente, com uma interface em linha de comandos, que comunica com a API (ver figura 2.5). O servidor e cliente podem correr na mesma máquina ou então pode-se utilizar o cliente para ligar a *daemons* remotos. Estes podem também ser controlados por aplicações externas, através de chamadas à interface CLI ou diretamente à API.

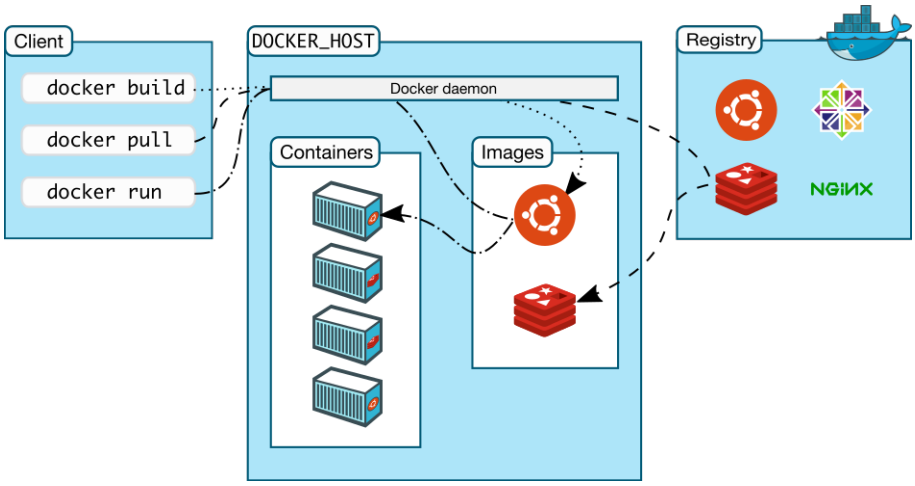

Figura 2.5: Arquitetura do Docker [**?**]

Outra componente importante do ecossistema Docker são os *registries*. Estes contêm imagens pré-configuradas, que podem de forma simples ser descarregadas e executadas. Estas imagens são construídas recorrendo a um ficheiro de configuração (denominado *Dockerfile*), onde estão especificadas as instruções para a construção dessa imagem, tipicamente recorrendo a outra como base.

Apesar da elevada adoção deste *container engine*, têm vindo a surgir outros com algumas diferenças na abordagem a este tipo de solução. Uma delas é o rkt, uma plataforma desenvolvida pela CoreOS.

A principal diferença entre o Docker e o rkt assenta nas suas arquiteturas. O rkt não utiliza um *daemon* para lançar os *containers*, sendo estes diretamente instanciados pela aplicação cliente, aquando dos comandos do utilizador. Os criadores desta plataforma referem que esta abordagem é mais compatível com sistemas tradicionais de *init*[1].

Existem outras implementações que devido à sua baixa maturidade ou falta de suporte não serão abordadas.

2.4 Orquestradores

A utilização de *containers* permite a decomposição de aplicações em pequenos módulos individuais. Se por um lado esta granularidade pode trazer grandes vantagens, por outro cria novos desafios na sua gestão, principalmente quando consideramos a escala a que os sistemas podem operar. Como exemplo, a Google lança 2000 milhões de *containers* por semana, ou pouco mais de 3000 por segundo. Em prática, quando é necessário administrar um *cluster*, é impossível lidar manualmente com problemas como gestão de configurações, distribuição de carga (*load balancing*) ou tolerância a falhas. É então necessária a utilização de plataformas que automatizem este processo, os chamados orquestradores. De seguida vamos analisar algumas destas plataformas.

[1] Processo base de um sistema operativo Unix que carrega todos os outros processos

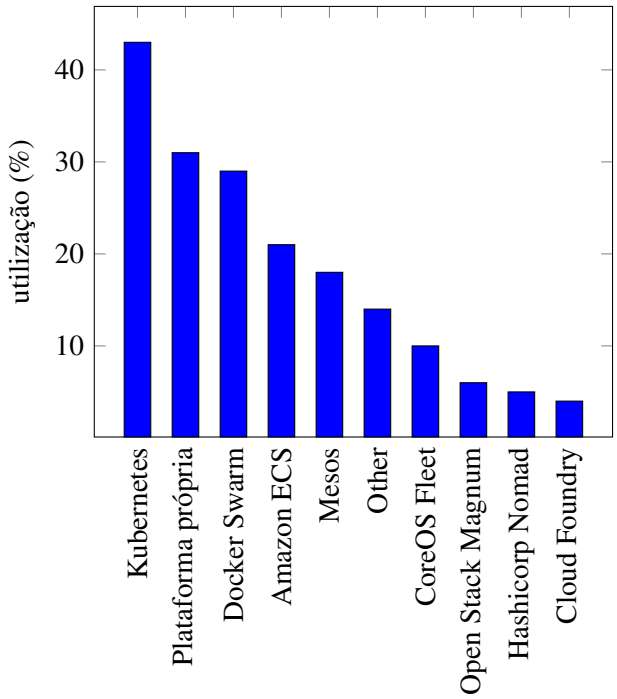

Figura 2.6: Orquestradores mais utilizados (sondagem [**?**])

2.4.1 Kubernetes

O Kubernetes é uma plataforma criada pela Google e que é atualmente gerida pela Cloud Native Computing Foundation. É neste momento a plataforma de orquestração de *containers* mais popular (ver gráfico 2.6), sendo um dos projetos mais ativos na comunidade *open-source*.

Por forma a simplificar o processo de operação de um *cluster*, o Kubernetes providencia então três funções principais:

- **Escalonamento**

  Os escalonadores têm a função de decidir onde instanciar os *containers*, de acordo com os recursos disponíveis. É possível especificar requisitos de CPU ou memória, de forma a que o escalonador mantenha uma contagem dos recursos utilizados e ainda disponíveis em cada *host* e possa assim garantir o correto funcionamento dos *containers*. Existe ainda desde a versão 1.5 a possibilidade de criar recursos opacos (denominados *opaque integer resources*), que permitem indicar, por exemplo, que certo módulo só pode ser instanciado num *host* que tenha determinado *hardware* ou periférico [**?**].

- **Descoberta de serviços e distribuição de carga**

  Em orquestração de *containers* pode haver um grande número de serviços, escalados horizontalmente. Dada a existência de múltiplas réplicas destes, torna-se necessário utilizar mecanismos que permitam detetá-los e aceder-lhes de forma dinâmica.

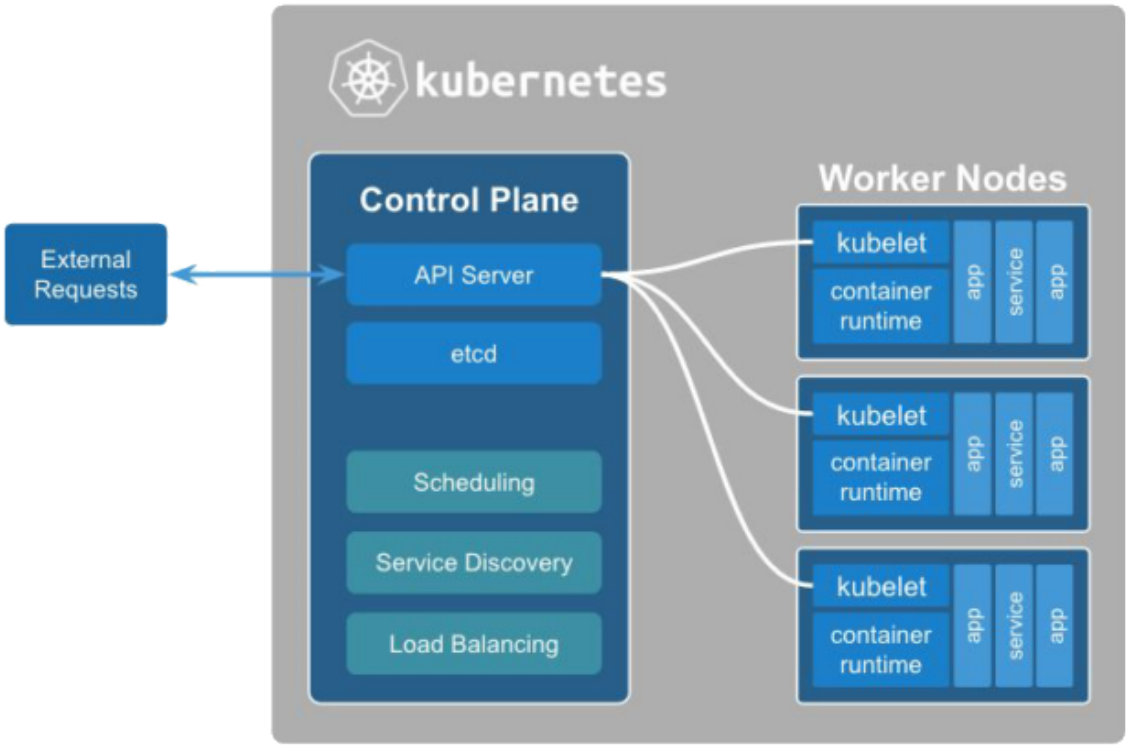

Figura 2.7: Arquitetura do Kubernetes [**?**]

O Kubernetes mantém registo de todos os serviços a correr no *cluster*, e providencia funcionalidades como distribuição da carga, coordenando e distribuindo o tráfego entre as réplicas disponíveis, e verificação do estado de funcionamento das mesmas.

- **Gestão de recursos**

  Diferentes tipos de aplicação podem consumir recursos de formas muito distintas, visto que algumas tarefas podem requerer mais CPU e outras mais memória, por exemplo. Assim, o Kubernetes tenta escalonar as tarefas de forma a não sobre-utilizar certos recursos, o que poderia causar instabilidade no *host*.

Segue-se uma explicação da arquitetura do Kubernetes. Como podemos ver na figura 2.7, esta plataforma é composta por um *control plane* e por nós *worker*. Um ou mais nós *master* providenciam o *control plane*, que é composto por certos componentes fundamentais na tomada de decisões e coordenação do *cluster*. Estes nós também podem instanciar *containers* dos utilizadores, mas tipicamente ficam reservados para estas tarefas de gestão. Os principais componentes do *control plane* são [**?**]:

- **kube-apiserver**

  Expõe a API do Kubernetes, a partir da qual são feitos pedidos ao *control plane*.

- **etcd**

  Base de dados chave-valor distribuída que armazena os dados operacionais do *cluster*.

- **kube-controller-manager**

  Corre *controllers*, que são responsáveis por tarefas de rotina como detectar falhas de nós ou *containers*, respondendo de forma apropriada.

- **kube-scheduler**

  Escalonador cuja tarefa é selecionar o nó mais apropriado para instanciar novos *containers*.

- **DNS**

  O Kubernetes corre um DNS interno como forma de providenciar *service discovery* através de nomes.

Os nós *worker* correm aplicações ou serviços dos utilizadores da *cloud*. Existem alguns componentes comuns aos dois tipos de nós:

- **kubelet**

  Principal componente de um nó em Kubernetes. É responsável por ficar à escuta de comandos por parte do *kube-apiserver* e desempenhar as ações solicitadas.

- **kube-proxy**

  Permite o encaminhamento de pedidos para os serviços correspondentes.

- **container engine**

  Tipicamente Docker, responsável pela execução dos *containers* propriamente ditos.

Um "objeto" em Kubernetes é uma entidade persistente que define algo, como uma aplicação e os seus componentes, políticas e argumentos de configuração, recursos disponíveis para as mesmas, etc. [**?**] Estes são definidos através de documentos YAML ou JSON. A plataforma utiliza estas definições para representar o estado desejado do *cluster*, e procura sempre mantê-lo. Alguns dos mais comuns são:

- **Pod**

  Unidade base do Kubernetes, é um *wrapper* à volta de um só *container* ou de um pequeno grupo destes que coexistem no mesmo espaço e com os mesmos recursos. Uma *Pod* é uma entidade efémera, que raramente é instanciada manualmente, mas sim por controladores. Representa uma instância de uma aplicação ou de módulos desta, e providencia recursos como rede e acesso a volumes de armazenamento aos *containers* que nela correm (ver figura 2.8).

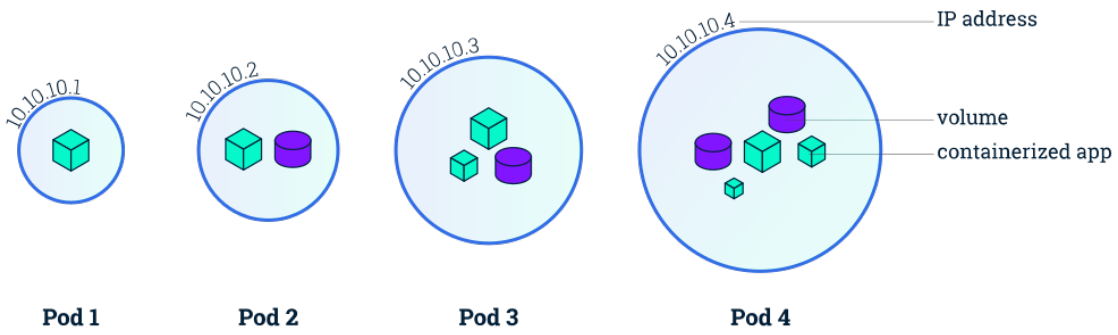

Figura 2.8: Kubernetes - Pods [**?**]

- **Service**

  Dada a efemeridade das Pods e a possibilidade da existência de várias réplicas das mesmas, cada uma com diferentes endereços de IP e localizadas em nós distintos, torna-se necessária a existência de uma entidade que faça a ligação entre os pedidos e as instâncias de *Pods* que vão responder a estes. Estas entidades chamam-se *Services* em Kubernetes (ver figura 2.9).

  Existem diferentes tipos de serviços, que variam na forma a como são acedidos:

  - **ClusterIP** (default) - Expõe o serviço num IP interno, apenas acessível a partir do interior do *cluster*.
  - **NodePort** - Expõe o serviço na mesma porta em todos os nós do *cluster* usando NAT, tornando-os acessíveis a partir do exterior.
  - **LoadBalancer** - Cria um *node balancer* externo na *cloud* utilizada (quando disponível) e atribui um IP externo fixo ao serviço.
  - **ExternalName** - Faz uso do módulo de DNS para expor o serviço usando um nome à escolha, através de registos CNAME.

  Os serviços utilizam tipicamente *labels* para se associarem às *Pods* correspondentes.

- **ReplicaSet**

  Esta entidade tem como função lançar e manter disponíveis um número *n* de réplicas de uma pod. Isto significa lançar novas réplicas quando estas falham.

- **Deployment**

  Apesar de ser possível lançar ReplicaSets de forma independente, a abordagem recomendada é utilizar Deployments, uma entidade que, para além de ser responsável por criar ReplicaSets, é mais flexível, permitindo aplicar alterações declarativas aos módulos, mantendo o estado e permitindo desfazê-las se desejado.

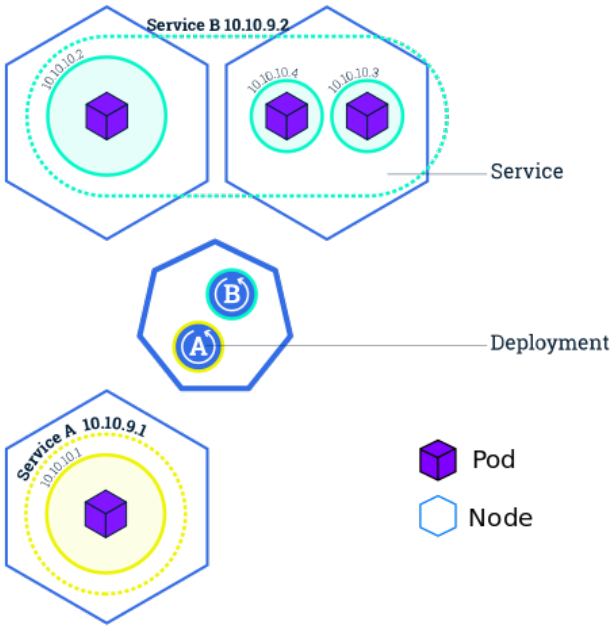

Figura 2.9: Kubernetes - Services [**?**]

2.4.2 Docker swarm mode

A plataforma *Docker* dispõe de um conjunto de funcionalidades de orquestração e gestão de *cluster*. Estas eram anteriormente parte de um sistema independente, mas foram integradas no *Docker engine* na versão 1.12.

Esta plataforma é gerida diretamente a partir da interface do Docker, não sendo necessário instalar software adicional. Existe assim uma menor curva de aprendizagem com esta solução de orquestração, sendo que o modo de interação com o *cluster* é mais semelhante à utilização do Docker no modo normal.

Um *swarm* é um conjunto de nós que correm serviços. Uma máquina que esteja a correr o *Docker engine* pode entrar neste modo criando ou juntando-se a um *swarm* já existente.

Existem dois tipos de nós (ver figura 2.10):

- **Manager nodes** são nós que coordenam o *cluster* e distribuem tarefas

  Estes nós recorrem a uma implementação do algoritmo de consenso distribuído Raft [**?**] para manter um estado consistente do *swarm* e dos serviços que nele estão a correr. Para além disto servem a API REST utilizada para a gestão do *swarm*. Por defeito estes nós também funcionam como *worker nodes*, sendo no entanto possível desativar este comportamento.

- **Worker nodes** são os nós que executam as tarefas

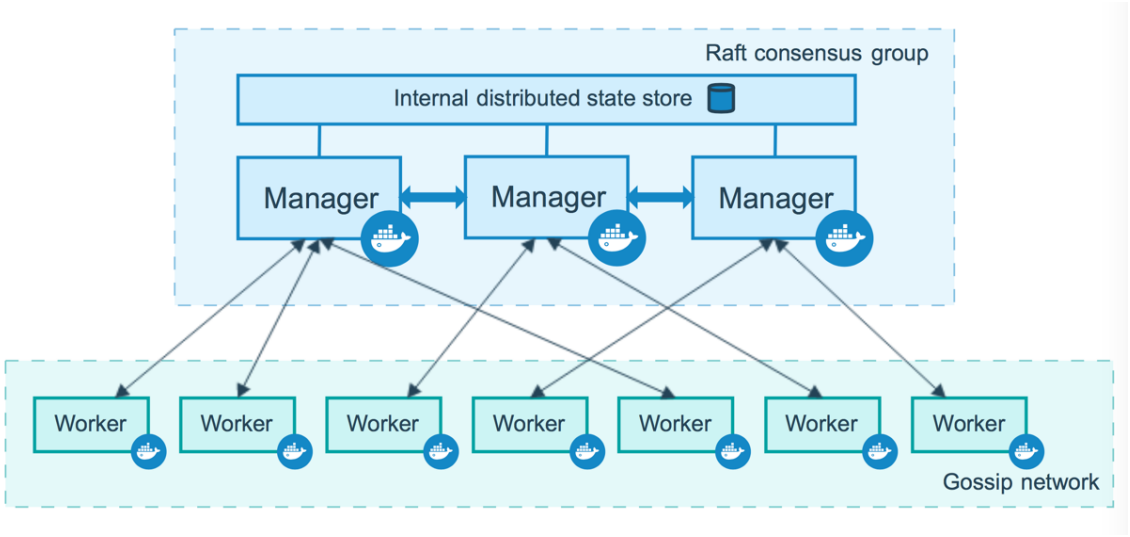

Figura 2.10: Swarm - Nodes [**?**]

Este tipo de nós não participa nas tarefas de gestão do *swarm*, sendo que apenas executam *containers* dos utilizadores.

No mínimo, um *swarm* é composto por um *manager node*. Uma só máquina pode correr vários nós de ambos os tipos, mas em cenários de produção tipicamente distribui-se a carga por várias máquinas. Assim é possível haver tolerância a falhas, sendo que o *swarm* consegue recuperar da falha de $(N-1) \div 2$ *manager nodes*, em que N é o número de nós deste tipo [**?**]. É recomendado um máximo de sete destes nós por *swarm*.

Serviços em *swarm* são definições de conjuntos de tarefas a executar em nós do tipo *worker*. É através destes que um utilizador pode instanciar *containers* no *cluster*. Num serviço é possível configurar parâmetros de execução como portas de acesso, número de réplicas a executar, limites e reserva de recursos, etc.

Existem dois tipos de serviço:

- **Replicated services** são serviços para os quais está definido um número de réplicas de uma tarefa, as quais serão executadas nos *worker nodes* disponíveis. Os *manager nodes* mantêm o estado desejado, lançando réplicas extra para substituir outras cuja execução possa ter falhado. É através deste tipo de serviços que se podem utilizar as funcionalidades de *load balancing*, distribuindo a carga pelas diferentes réplicas instanciadas.

- **Global services**, por sua vez, definem tarefas que devem ser executadas em todos os nós. Haverá sempre uma cópia por cada nó, sendo automaticamente instanciadas para novos nós que se juntem ao *swarm*.

A figura 2.11 mostra um exemplo de um *replicated service* configurado para três réplicas e um *global service*.

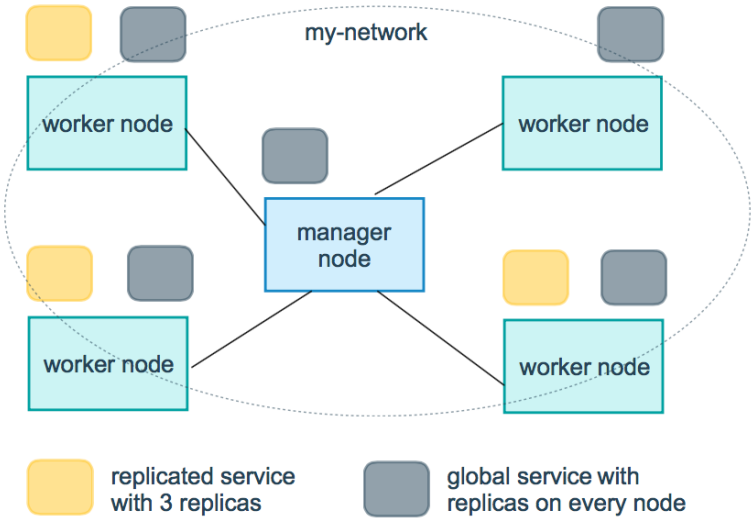

Figura 2.11: Swarm - Service types [**?**]

As tarefas são as unidades atómicas de um *swarm*, sendo que a sua função é a instanciação de um *container*. Estas têm estados associados, e o orquestrador tenta sempre manter o estado desejado especificado pelo serviço. Por exemplo, se o estado desejado de uma tarefa for **running** e a sua execução falhar, esta será automaticamente apagada e criada uma nova para a substituir.

A figura 2.12 mostra o ciclo de vida de um serviço, desde que é feito um pedido para a sua criação até às tarefas serem executadas num *worker node*.

2.4.3 Apache Mesos

Certas *frameworks* como o *Hadoop* são já fortemente distribuídas e executadas em *clusters*, escalonando tarefas em nós. Contudo, para muitos casos, pode ser desejável correr múltiplas *frameworks* deste tipo no mesmo *cluster*, optimizando a sua utilização e permitindo a partilha de recursos entre estas, reduzindo assim a replicação desnecessária de informação.

O Mesos [**?**] foi criado com este caso em mente, utilizando a filosofia de definir uma camada mínima que permita a partilha de recursos entre as *frameworks*, deixando ao cargo destas o escalonamento. Assim, este orquestrador não oferece uma solução completa por si só, necessitando sempre de pelo menos uma destas plataformas.

A arquitetura do Mesos consiste num processo *master*, que faz a gestão dos *daemons slave* e das *frameworks*. Os *daemons slave* correm em cada um dos nós do *cluster* e as diferentes *frameworks* executam tarefas nesses nós. Uma visão geral desta arquitetura pode ser consultada na figura 2.13.

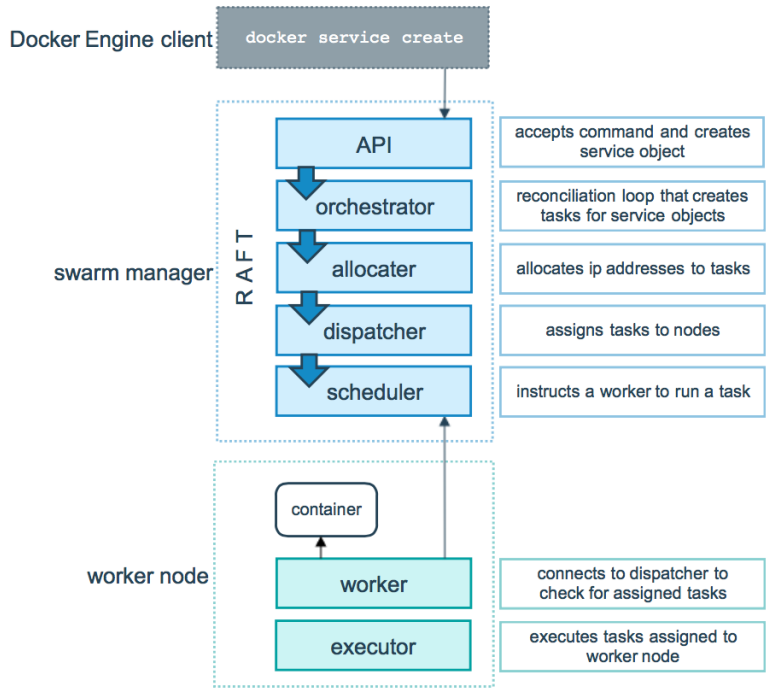

Figura 2.12: Swarm - Service lifecycle [**?**]

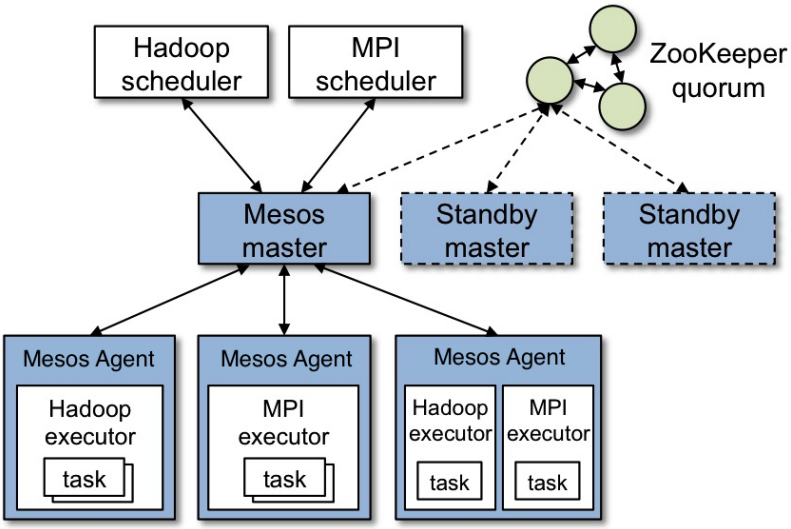

Figura 2.13: Mesos - arquitetura com duas *frameworks* (Hadoop e MPI) [**?**]

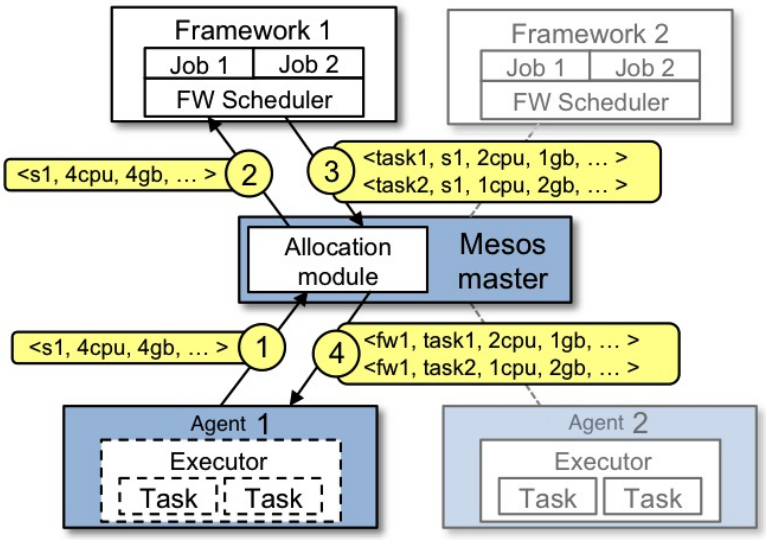

Figura 2.14: Mesos - alocação de recursos [**?**]

O processo *master* oferece recursos de computação às *frameworks*, que respondem com um conjunto de tarefas que querem correr, e de que forma os recursos disponibilizados devem ser alocados a cada uma delas. De seguida o *master* instancia as tarefas nos nós disponíveis (ver figura 2.14).

A plataforma utiliza o **Apache Zookeeper** para permitir tolerância a falhas no *master*, fazendo a eleição de um novo líder de entre os *masters* que se encontram inativos (*standby*). O isolamento de recursos entre *frameworks* é feito recorrendo a *containers*, havendo a possibilidade de utilizar para este efeito as seguintes opções [**?**]:

- **Mesos**, opção nativa de *containerização*, e segundo a documentação a solução preferida
- **Docker**, que pode ser utilizada quando uma aplicação necessitar de funcionalidades específicas desta solução
- **Composing**, uma solução híbrida que permite combinar ambas as soluções anteriores, definindo qual deve ser utilizada para cada tarefa

Uma das *frameworks* que pode ser usada em conjunto com o Mesos é o **Marathon** [**?**], uma plataforma de orquestração de aplicações em *containers*. Esta *framework* assume um papel de coordenador de todo o *cluster*, instanciando tanto aplicações como outras *frameworks* (que podem também lançar as suas próprias aplicações) e assegurando-se da sua disponibilidade.

2.5 Container Networking

Num ambiente descentralizado, em que diferentes serviços podem ser instanciados em vários *hosts*, potencialmente em locais fisicamente distantes, torna-se necessário garantir a conectividade

entre estes. Para além disto pode ser importante que os *containers* estejam isolados em diferentes redes virtuais.

Existem duas principais especificações para redes de *containers*: Container Network Model (**CNM**) e Container Network Interface (**CNI**).

CNM é um modelo especificado pelo Docker em que são utilizados *drivers* associados a redes virtuais. A implementação principal deste modelo é o **libnetwork**, do próprio Docker. Esta disponibiliza *drivers* nativos como *bridge* ou *overlay*.

CNI é uma especificação e conjunto de bibliotecas para o desenvolvimento de plugins que disponibilizem conectividade de rede entre containers. Existem várias implementações desta especificação, sendo que três das mais utilizadas são o Weave Net, o Calico e o Flannel.

Na tabela 2.1 comparamos algumas das soluções encontradas a nível de *container networking*. Segue-se uma breve explicação de cada parâmetro comparado.

Tabela 2.1: Container Networking

Funcionalidade	**Docker Overlay Network**	**Calico**	**Flannel**	**Weave Net**
Open Source	Sim	Sim	Sim	Sim
Modelo de rede	Overlay VXLAN	L3 com encapsulamento opcional	Overlay VXLAN ou UDP	Overlay VXLAN ou UDP
Isolamento de aplicações	CIDR schema	Policy schema	CIDR schema	CIDR schema Profile schema
Protocolos suportados	Todos	Todos	Todos	Todos
Serviço DNS embutido	Sim	Não	Não	Sim
Load-balancer embutido	Sim	Não	Não	Não
Canal encriptado	Sim	Não	Não	Sim
Redes parcialmente conectadas	Não	Não	Não	Sim
vNIC por container	Sim	Sim	Não	Sim
Suporte a multicast	Não	Não	Não	Sim
CNI	Não	Sim	Sim	Sim
CNM	Sim	Sim	Não	Sim
Suporte a Openstack	N/A	Sim	Não	Não
Suporte a Kubernetes	N/A	Sim	Sim	Sim
Suporte a Apache Mesos	N/A	Sim	Sim	Sim
Suporte a Docker swarm mode	Sim	Sim	Não	Sim
Suporte a rkt	N/A	Sim	Sim	Sim

- **Open Source**: projeto de código aberto
- **Modelo de rede**: de que forma a solução cria a rede virtual entre os *containers*
- **Isolamento de aplicações**: esquema utilizado para o isolamento de aplicações na rede virtual. Pode ser baseada em diferentes sub-redes (CIDR), políticas ou perfis de acesso.
- **Protocolos suportados**: quais os protocolos IP suportados. As soluções comparadas suportam todos os protocolos.
- **Serviço DNS embutido**: a solução providencia um serviço de resolução de nomes.
- **Load-balancer embutido**: a solução providencia um serviço de distribuição de carga. Não existindo, esta pode ficar a cargo do orquestrador.
- **Canal encriptado**: é possível configurar encriptação na comunicação dentro da rede virtual
- **Redes parcialmente conectadas**: a solução suporta redes em que os nós não estão todos conectados entre si, encaminhando o tráfego pelos nós disponíveis
- **vNIC por container**: os *containers* têm mais uma camada de abstração, sendo que cada um tem uma interface de rede virtual separada
- **Suporte a multicast**: suporte a comunicações *multicast* entre *containers* na rede virtual
- **CNI**: implementa a especificação Container Network Interface
- **CNM**: implementa a especificação Container Network Model
- **Suporte a Openstack**: a solução suporta a *framework* Openstack
- **Suporte a Kubernetes**: suporte ao orquestrador Kubernetes
- **Suporte a Apache Mesos**: suporte ao orquestrador Apache Mesos
- **Suporte a Docker swarm mode**: suporte às ferramentas de orquestração do Docker
- **Suporte a rkt**: suporte ao *container engine* rkt

2.6 Redes de computadores

Um dos entraves mais claros à produção televisiva virtual passa pela relação custo / benefício, nomeadamente das ligações de rede [**?**]. Dada a prática comum de utilizar vídeo não comprimido para obter qualidade comparável aos meios tradicionais de transmissão (i.e. cabos SDI) [**?**], o hardware de rede mais comum mostra-se insuficiente. Como exemplo, uma só *stream* de vídeo não comprimido Full HD a 50 fotogramas por segundo necessita de cerca de 2Gb/s, sendo expectável que num cenário de produção de um programa de televisão existam múltiplas câmaras, acompanhadas de microfones e outros equipamentos.

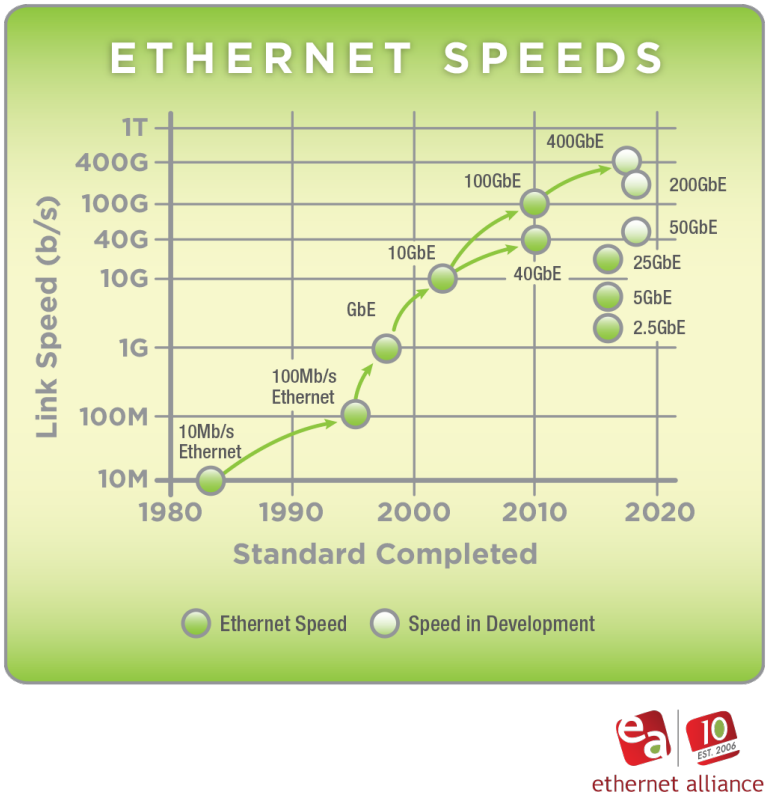

Figura 2.15: Evolução na velocidade das redes ethernet [**?**]

No entanto têm-se vindo a verificar desenvolvimentos em tecnologia de redes de vários tipos (fibra, *ethernet*) com capacidade para transmitir informação em velocidades na ordem das dezenas ou até centenas de gigabits por segundo [**?**] (ver figura 2.15). Como tal, começa a ser possível produzir conteúdo televisivo de alta qualidade na *cloud*, e é previsível que o custo do equipamento necessário continue a baixar e a acompanhar as evoluções na indústria televisiva.

2.6.1 Métodos de difusão de pacotes

Em redes de computadores, uma comunicação pode seguir diferentes abordagens no que diz respeito à difusão de pacotes (ver figura 2.16), consoante o número de recipientes destes. Numa comunicação *unicast*, um pacote tem como destino apenas um nó na rede. No modelo *broadcast* há vários receptores, sendo que a comunicação é feita para todos os endereços numa determinada sub-rede.

No caso de comunicações *multicast* [**?**] existem também vários receptores, mas há um maior controlo sobre estes. Baseia-se numa lógica de subscrições, em que nós interessados em receber determinados pacotes se juntam a um grupo, sendo que para este efeito devem enviar uma mensagem *join* para a rede, utilizando o protocolo *IGMP*. Assim que deixem de estar interessados em receber pacotes deste grupo, devem enviar uma mensagem *leave*. O emissor das mensagens

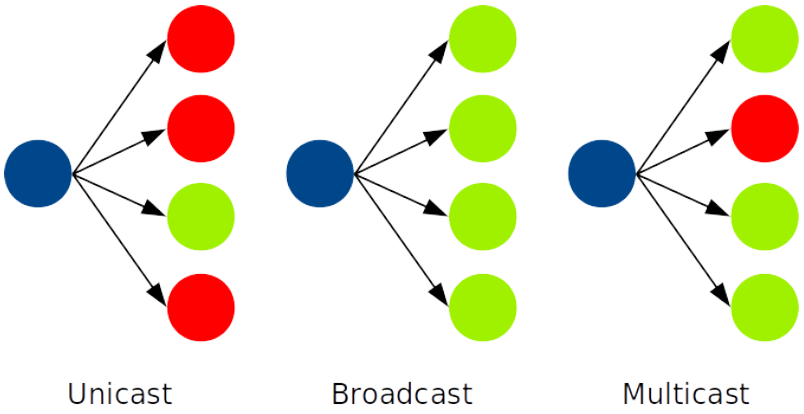

Figura 2.16: Unicast vs Broadcast vs Multicast

não necessita de conhecer os nós interessados, dado que esta gestão é feita ao nível do *router* ou *switch*.

2.7 Conclusões

Neste capítulo foi revisto o estado da arte sobre computação na *cloud*, formalizando a sua definição e explorando alguns dos conceitos e modelos que a definem. Foi abordada a área de DevOps, sobre a qual foram descritas algumas das suas responsabilidades e tipos de ferramentas que normalmente utilizam para a gestão de *clouds*.

De seguida, foi feita uma análise sobre dois tipos de virtualização: hipervisores e *containers*. Foi feita uma comparação entre estas tecnologias e foram explicados alguns dos seus componentes mais importantes. A análise da tecnologia de *containers* foi mais detalhada devido à sua maior aplicabilidade ao caso em estudo.

Quanto aos orquestradores, foi explicada a sua importância na gestão de uma *cloud*, tendo sido de seguida exploradas algumas das mais importantes plataformas nesta área. Foi também feita uma comparação entre diferentes soluções de *container networking*. De forma mais geral, foi apresentada a melhoria progressiva da velocidade em redes de computadores e a importância deste facto no contexto dos estúdios televisivos virtuais.

Foi decidido utilizar **Docker**, pela sua grande aceitação, suporte e por ser a tecnologia de base do trabalho anteriormente desenvolvido no cenário de teste.

Quanto ao orquestrador, optamos pelo **Kubernetes**, porque apesar da maior dificuldade de configuração e utilização comparativamente por exemplo ao Docker *swarm mode*, é um projeto em desenvolvimento muito ativo e em grande crescimento, para além de muito completo a nível das suas potencialidades, permitindo assim uma melhor adaptação a um cenário mais genérico. O Apache Mesos foi descartado pela sua baixa aplicabilidade ao cenário de teste.

A nível de sistema operativo, devido ao facto de os *containers* já disponíveis para o cenário de testes terem sido desenvolvidos para ambiente Linux, iremos utilizar uma distribuição deste

sistema operativo. Foi selecionado o **CentOS**, pelo seu suporte à ferramenta **kubeadm** de configuração do orquestrador, pela sua elevada estabilidade e pela grande adoção desta distribuição a nível empresarial.

Finalmente, a solução de *container networking* selecionada foi o **Weave Net**, devido ao seu suporte a comunicações *multicast*, que era fundamental ao funcionamento do cenário de teste utilizado. Para além disso, esta é simples de instalar e configurar no ambiente de orquestração do Kubernetes.

Capítulo 3

Protótipo de estúdio televisivo virtual

Neste capítulo vamos explicar o cenário de teste utilizado, nomeadamente a sua arquitetura e funcionalidades.

Este assenta sobre o trabalho previamente realizado na MOG Technologies por Miguel Poeira [**?**], que propõe uma aplicação distribuída na *cloud* cuja função é a recepção e retransmissão de múltiplas *streams* de conteúdo televisivo através de redes IP, permitindo a um realizador comutar entre estas, escolhendo assim o conteúdo de saída final. A utilização da *cloud* potencia a escalabilidade da aplicação, de forma a lidar com tarefas como o *transcoding* dos sinais em tempo real e a sua comutação, tarefas tradicionalmente feitas em hardware dedicado.

3.1 Arquitetura

Esta aplicação tem uma arquitetura baseada em "blocos" (figura 3.1) instanciados em *containers* Docker, baseados na imagem Ubuntu 14.04. Os sinais são descomprimidos à entrada, de forma a manter a qualidade esperada nos métodos tradicionais de produção, sendo novamente comprimido à saída.

Para este efeito, a aplicação recorre à arquitetura de referência JT-NM [**?**], que estabelece um conjunto de conceitos sobre o funcionamento de módulos ou *Nodes*, interligados entre si por redes Ethernet. Existem dispositivos na rede que disponibilizam fontes (*Sources*) de conteúdo de diversos tipos, como áudio, vídeo, entre outros. A uma *stream* destes conteúdos dá-se o nome de *Flow*. A cada unidade destes *Flows* dá-se o nome de *Grain*, que representa um elemento indivisível de uma *stream* como uma *frame* de vídeo ou uma *sample* de áudio.

São ainda utilizadas as normas do NMOS (Networked Media Open Specification) [**?**], que propõem um modelo para o funcionamento e a interoperabilidade dos *Nodes* especificados pela arquitetura do JT-NM. Estes devem expor APIs REST que lhes permitem receber ordens e tomar as ações necessárias. Estas normas propõe também soluções para outros tipos de problemas que

surgem na implementação da arquitetura de referência, como o suporte a diferentes protocolos de vídeo ou a sincronização de *streams*.

Os módulos definidos na aplicação já existente são:

- **Input Distributor**:

  Recebe uma *stream* MPEG-TS transportada por RTP à entrada, descomprime-a, separa os sinais de áudio e vídeo (*demux*) e retransmite-os através de *multicast*, utilizando a norma RFC 4175 [**?**] (transmissão de vídeo não comprimido por RTP).

- **Proxy Transcoder**:

  Recebe do canal *multicast* um par de sinais áudio e vídeo, comprimindo-os e fornecendo através de MPEG-DASH uma versão de baixa resolução (*proxy*) dos mesmos, permitindo ao realizador fazer a pré-visualização dos conteúdos, tanto à entrada (Input Distributors) como à saída (Output) da aplicação.

- **Video Switcher**:

  Subscreve vários endereços de *multicast*, correspondentes às várias *streams* de entrada, e implementa um *buffer* em memória da informação que vai recebendo. Recebe da Business Logic um comando para comutar entre as *streams*, encaminhando os sinais selecionados para um endereço *multicast* diferente do original.

- **Business Logic**:

  Este módulo seria responsável pela comunicação com o exterior, implementando a lógica de negócio, recebendo comandos e disponibilizando informações da aplicação, através de uma API REST. No entanto, à data da elaboração deste trabalho, este módulo não se encontrava ainda implementado, pelo que não faz parte do cenário de teste.

- **Output**:

  O módulo final da cadeia, que recebe os sinais de áudio e vídeo não comprimido por *multicast*, comprime-os e envia-os através de uma *stream* MPEG-TS. A saída deste módulo pode alimentar a entrada de um Input Distributor, criando assim um sistema em cascata.

3.2 Limitações

Apesar deste protótipo ter sido desenvolvido tendo em vista os conceitos de *cloud computing*, a instanciação dos seus módulos no ambiente da empresa era ainda feita de forma pouco automatizada.

Estava a ser utilizada a ferramenta *docker-compose*, que permite a configuração de aplicações com múltiplos *containers* Docker. No entanto esta não tirava proveito das múltiplas máquinas disponíveis na MOG, e a transição para um cenário distribuído requeria algum trabalho a nível da configuração da *cloud* e ambiente de orquestração.

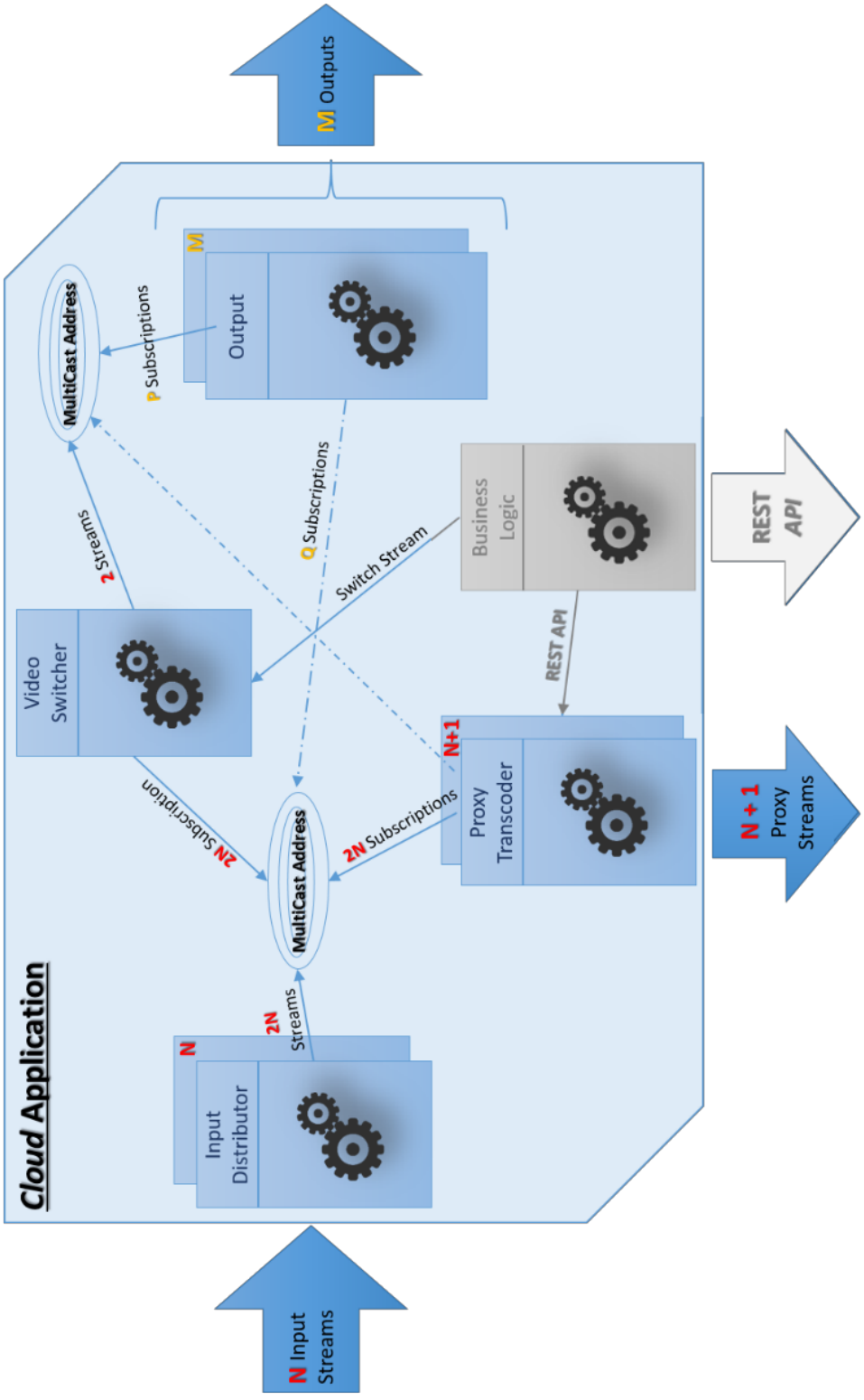

Figura 3.1: Arquitetura da aplicação do cenário de teste [**?**]

Conforme anteriormente referido, procuramos colmatar estas limitações através da implementação da solução proposta, que será descrita com mais detalhe no capítulo 4. Esta procura ter aplicabilidade a vários tipos de aplicações, não se cingindo a este cenário de teste.

3.3 Aplicação ao projeto

Conforme referido no capítulo 1, a solução proposta nesta dissertação utiliza como caso de teste e validação o cenário dos estúdios televisivos virtuais.

O cenário utilizado como referência para a implementação deste trabalho consiste em duas *streams* de entrada, existindo assim dois *Input Distributors*, três *Proxy Transcoders*, um *Video Switcher* e um *Output*, sendo assim testados todos os módulos implementados até agora.

Este caso procura simular a cobertura de um evento onde existem duas câmaras de vídeo cujos sinais não comprimidos são transmitidos por comunicação *multicast* pelos *Input Distributors*. Os *Proxy Transcoders* recebem esses sinais e geram versões de pré-visualização, que são recebidas num estúdio de produção remoto. Aí encontra-se um realizador, que pode comutar entre os sinais das duas câmaras. Esta funcionalidade é disponibilizada pelo *Video Switcher*, que recebe as *streams* e comuta entre elas, transmitindo a selecionada por *multicast*. Finalmente, o *Output* comprime a *stream* escolhida e transmite-a para uma entidade responsável pela difusão do sinal.

Capítulo 4

Descrição do problema e solução proposta

Neste capítulo iremos descrever o problema e a implementação de uma plataforma para a instanciação de estúdios televisivos virtuais numa *cloud* privada. Serão abordados os passos tomados para a configuração da infraestrutura necessária, bem como alguns detalhes sobre o hardware informático utilizado. Serão também apresentados resultados de alguns testes de desempenho de rede, de modo a confirmar a viabilidade da transmissão de conteúdo televisivo nesta infraestrutura.

Quanto ao software desenvolvido, serão expostos os pontos e decisões mais relevantes a nível de arquitetura, detalhes de implementação, comunicação com a interface e monitorização.

Tentaremos ainda demonstrar a aplicabilidade da solução encontrada a outros tipos de sistemas, para além dos estúdios televisivos.

4.1 Descrição do problema

O problema que foi descrito no início do projeto passava por encontrar uma solução que permitisse a simplificação e automatização de alguns processos da instanciação de aplicações numa *cloud*, mais propriamente de estúdios televisivos virtuais.

Do ponto de vista deste trabalho, foi especificado que deveria ser configurada uma *cloud*, inicialmente com duas máquinas, que formassem um *cluster*. Acima destas, deveria haver uma camada (*deployer*) que recebesse um ficheiro (XML ou JSON), com uma estrutura a definir, o qual descreveria o cenário de um estúdio televisivo a ser instanciado em *containers*, com base nos módulos previamente implementados [**?**]. Este ficheiro deveria também ser validado pela aplicação, para garantir que não possuía erros e que cumpria a estrutura especificada. Finalmente, o cenário deveria ser instanciado e métricas do seu funcionamento deveriam ser recolhidas. Esta arquitetura básica pode ser vista na figura 4.1.

Os requisitos de uma solução para este problema são (ver tabela 4.1):

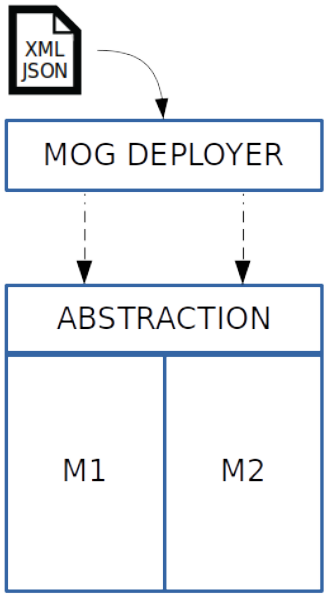

Figura 4.1: Arquitetura inicial da aplicação

- **Cloud privada (R001)**:

  Deve ser configurada uma *cloud* privada que permita a instanciação de um estúdio televisivo virtual em *containers*.

- **Orquestrador (R002)**:

  Deve ser configurada uma solução de orquestração que ajude à gestão da *cloud* e dos *containers* nela instanciados.

- **Comunicação multicast (R003)**:

  Os *containers* instanciados nesta aplicação devem poder utilizar comunicação multicast para comunicação entre eles, de forma a permitir o *deployment* do cenário de teste previamente discutido.

- **Definição de formato do cenário (R004)**:

  Deve ser definido um formato que descreva o cenário do estúdio televisivo virtual.

- **Recepção do cenário (R005)**:

  Deve ser desenvolvida uma aplicação que receba um ficheiro que represente o cenário, com todas as informações e configurações necessárias para a sua instanciação na *cloud*.

- **Validação do cenário (R006)**:

  A aplicação deve permitir fazer a validação do cenário recebido, retornando o resultado da mesma.

Tabela 4.1: Definição do problema - requisitos

ID	Nome	Prioridade
R001	Cloud privada	Fundamental
R002	Orquestrador	Fundamental
R003	Comunicação multicast	Fundamental
R004	Definição de formato do cenário	Fundamental
R005	Recepção do cenário	Fundamental
R006	Validação do cenário	Fundamental
R007	Instanciação do cenário	Fundamental
R008	Reserva de recursos	Importante
R009	Containerização	Desejável
R010	Métricas e monitorização	Desejável

- **Instanciação do cenário (R007)**:

  Dado a recepção de um cenário válido, a aplicação deve ser capaz de o instanciar em *containers* na *cloud*.

- **Reserva de recursos (R008)**:

  A aplicação deve conseguir reservar os recursos necessários para cada módulo no orquestrador, nomeadamente de largura de banda.

- **Containerização (R009)**:

  Idealmente a solução estará ela própria instanciada num *container*, na *cloud*.

- **Métricas e monitorização (R010)**:

  Deverão ser fornecidos à interface métricas de utilização da *cloud* e uma solução de monitorização do estado da mesma.

4.2 Tecnologias utilizadas

A tabela 4.2 apresenta as tecnologias, linguagens de programação e bibliotecas utilizadas na implementação desta solução. As mais relevantes são detalhadas nas secções correspondentes neste capítulo e / ou no capítulo 2.

4.3 Cloud privada

Apesar da maior simplicidade em utilizar um provedor de serviços de *cloud computing*, foi decidido montar uma *cloud* privada nas instalações da MOG, visto que existiam máquinas disponíveis para este efeito e que o acesso mais direto a estas permitia testar diferentes configurações de *hardware* e conectividade de rede e as suas influências no cenário. Foram utilizadas sete máquinas

Tabela 4.2: Tecnologias utilizadas

Sistema operativo		CentOS 7
Container engine		Docker 17
Orquestrador		Kubernetes 1.6
Especificação de formato		JSON Schema
Container networking		Weave Net
Testes de performance de rede		iperf3
Template engine		Mustache
Programação da aplicação	Linguagem	Python 3
	Bibliotecas	jsonschema
		Pystache
		Flask
		Kubernetes Python Client
Monitorização	Recolha	cAdvisor
		Heapster
	Armazenamento	InfluxDB
	Visualização	Grafana

físicas para a criação do *cluster*, cujas especificações técnicas podem ser encontradas no anexo A. A figura 4.2 mostra a infraestrutura de rede e hardware utilizada.

4.3.1 Sistema operativo

Como primeiro passo foi necessário instalar um ambiente de base idêntico em todas as máquinas que permitisse fazer o *deployment* da plataforma de orquestração escolhida, o Kubernetes. Devido à complexidade da configuração do orquestrador em causa, conforme já referido no capítulo 2, os responsáveis pelo seu desenvolvimento criaram a ferramenta kubeadm, que permite a automação de muitas das tarefas de instanciação de um *cluster*. Esta ferramente é, à data da escrita deste documento, apenas compatível com Ubuntu 16.04, CentOS 7 ou HypriotOS 1.0.1 ou superior. Assim, foi feita uma instalação base da distribuição **CentOS 7**. Esta foi escolhida devido ao seu bom suporte e popularidade a nível empresarial e à sua estabilidade.

Antes da instanciação do orquestrador, foi necessário instalar em cada máquina a versão mais recente do **Docker** engine, kubernetes e kubeadm, ferramentas estas disponíveis nos respetivos repositórios oficiais. Foram também feitas as configurações de rede, nomeadamente o acesso à rede interna de 10 Gb, que é feito utilizando a especificação **LACP** que prevê a agregação de múltiplas *interfaces* de rede [**?**].

4.3.2 Instanciação do orquestrador

Para a instanciação do orquestrador escolhido, o **Kubernetes**, foi utilizada a ferramenta **kubeadm**. Esta tem a limitação de criar *clusters* apenas com um nó *master*, o que, para o caso de teste, se mostrou suficiente.

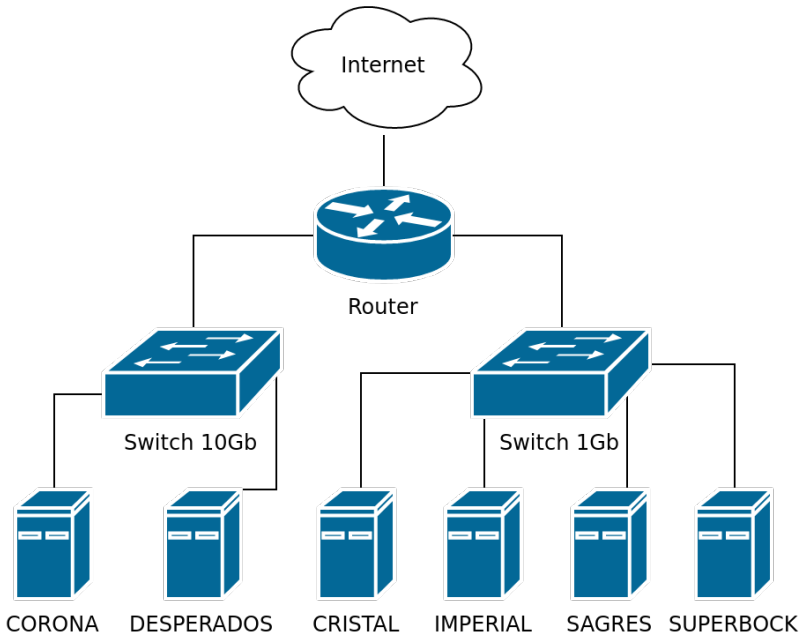

Figura 4.2: Infraestrutura de rede e hardware

Os passos para esta instanciação passam primeiramente por inicializar o nó *master* numa máquina a escolha. É devolvido pela ferramenta um *token* que permite aos restantes nós juntarem-se ao *cluster*. De seguida, é aplicada a Container Network Interface escolhida, neste caso o **Weave Net**. Finalmente são inicializados todos os nós *worker* que se queiram juntar, fornecendo ao kubeadm o *token* criado pelo *master* juntamente com o seu endereço IP e porta.

Os *containers* a utilizar no cenário de testes estavam disponíveis num *Docker registry* privado. O acesso a este foi configurado no *Docker engine* de todas as máquinas.

4.3.3 Desempenho da rede

Dado que os cenários seriam instanciados em *containers*, tornou-se importante verificar se o *overhead* acrescentado pela *Container Network* não se tornaria impeditivo ao bom funcionamento dos módulos. Para isso foi utilizada a ferramenta **iperf3**, que permite testar a largura de banda, perdas de pacotes e outras métricas da comunicação entre máquinas do *cluster*.

Foram feitos testes nas duas máquinas que, à data da escrita do documento, estavam equipadas com interfaces de rede de 10Gbit e em duas máquinas com interfaces tradicionais de 1Gbit (ver anexo A para mais detalhes sobre as suas especificações). Estas placas de rede suportam a funcionalidade *jumbo frames*, que permite aumentar o tamanho de uma *frame* ethernet do tradicional valor de 1500 bytes, imposto pelo *standard* IEEE 802.3 [**?**]. Dado que cada uma das *frames* possui um cabeçalho de tamanho fixo, ao aumentar a dimensão total reduz-se o *overhead* na transmissão de pacotes de grande dimensão, como pode ser desejável no caso da transmissão de conteúdo multimédia.

Tabela 4.3: Testes de performance de rede

Capacidade da ligação	Largura de banda	Comunicação	Percentagem
1Gbps	935Mbps	*Host-to-host*	93.5%
	903Mbps	*Container-to-container*	90.3%
10Gbps	9.42Gbps	*Host-to-host*	94.2%
	9.01Gbps	*Container-to-container*	90.1%

Os resultados destes testes são apresentados na tabela 4.3. Estes foram feitos na comunicação entre as máquinas físicas (*host-to-host*) e entre *containers* (*container-to-container*), utilizando o protocolo TCP. Foi medida a largura de banda conseguida e feita a comparação com a capacidade da ligação. É de ressalvar que nos testes em máquinas com interface de 10Gbps foi necessário utilizar a opção de paralelismo do iperf, visto que utilizando apenas um núcleo de processamento este atingia uma taxa de utilização de 100%, causando um *bottleneck*.

4.4 Panamax

A plataforma **Panamax** foi desenvolvida com o intuito de oferecer uma forma simples e intuitiva de um utilizador lançar aplicações modulares em *containers* num *cluster*. Esta plataforma consiste em serviços que expõem os recursos disponíveis numa *cloud* de forma a possibilitar a instanciação de aplicações. Os utilizadores interagem com a plataforma através de uma interface Web, e a comunicação entre esta e os serviços é feita através de pedidos REST, sendo os dados transmitidos no formato JSON. Foi definida uma especificação, baseada na tecnologia JSON Schema, para esta comunicação. Um dos requisitos consistia na monitorização do estado da *cloud* e das aplicações, sendo que foram disponibilizadas ferramentas para este efeito. Uma visão geral da arquitetura da aplicação pode ser vista na figura 4.3.

Para cumprir os requisitos definidos anteriormente (tabela 4.1), foi então necessário estabelecer alguns requisitos extra na solução proposta (tabela 4.4):

- **API REST (R101)**:

  Foi necessário definir uma API que disponibilizasse os serviços da aplicação à interface.

- **Preparação do cenário (R102)**:

  Foi necessário implementar uma camada que fizesse a preparação do cenário para instanciação, calculando os valores necessários para a reserva de recursos.

- **Transformação do cenário (R103)**:

  Dadas as diferenças entre o formato especificado para o cenário e o formato dos pedidos ao orquestrador, foi imperativo implementar uma camada que preparasse os ficheiros para a criação dos recursos.

- **Comunicação com o orquestrador (R104)**:

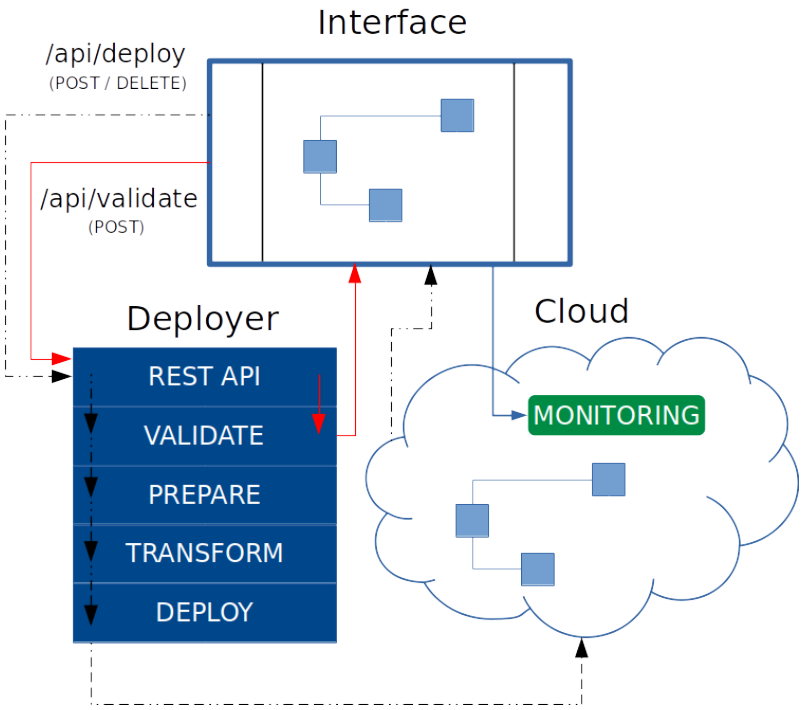

Figura 4.3: Arquitetura do Panamax

Foi necessário criar uma camada que comunicasse os pedidos de recursos ao orquestrador, recorrendo à API disponibilizada pelo mesmo, e retornasse o resultado da instanciação dos módulos à interface.

4.4.1 Tipos de módulos

No contexto desta plataforma, chamamos módulos aos componentes de um estúdio televisivo. Existem dois tipos de módulos:

- **Módulos dinâmicos** são módulos detectados de forma dinâmica no ambiente de rede, através da tecnologia **mDNS**. Estes são parte da aplicação, mas já estão previamente instanciados. Podem ser por exemplo câmaras ou microfones acessíveis pela rede, servidores de armazenamento de conteúdos, etc.

Tabela 4.4: Panamax - requisitos

ID	Nome	Prioridade
R101	API REST	Fundamental
R102	Preparação do cenário	Fundamental
R103	Transformação do cenário	Fundamental
R104	Comunicação com o orquestrador	Fundamental

- **Módulos estáticos** são os módulos da aplicação de teste previamente descrita. São configurados aquando do seu lançamento, que é feito na *cloud* privada.

Assim, apesar do formato JSON Schema estar preparado para módulos destes dois tipos, são mais relevantes para este trabalho os módulos estáticos, pois apenas estes são instanciados no *cluster*.

4.4.2 Formato do cenário

O **JSON Schema** é um formato para especificar a estrutura de informação em JSON. É utilizada para definir restrições, regras e descrições a um documento JSON. Esta especificação foi utilizada dada a necessidade de estabelecer um formato de comunicação entre a interface, que constrói o cenário e o *deployer*, que tem de o validar por forma a interpretá-lo e prepará-lo para instanciação na *cloud*. Desta forma, garante-se que a interface irá comunicar um cenário válido, passível de ser instanciado.

Para além da comunicação é utilizado para gravar o estado da interface, podendo ser exportado e importado para fazer posteriores alterações ao cenário.

Na raiz do *schema* existem três listas, uma para cada tipo de módulo e uma para as ligações entre eles.

As ligações contêm dois *endpoints*, o de origem e o de destino, sendo que cada um deles é composto pelo nome do módulo e do *pin* correspondente.

Os módulos estáticos são definidos por um nome, uma imagem de *container*, um conjunto de *pins* e por variáveis de ambiente. Os módulos dinâmicos têm também um nome e um conjunto de *pins*, contendo ainda a especificação do tipo de módulo (câmara, gerador de sinal, etc.) (ver figura 4.4).

Os *pins* previamente referidos descrevem pontos de transmissão de informação num módulo, sendo equivalentes a portas. Cada módulo possui duas listas, uma para *pins* de entrada e outra para *pins* de saída. Cada *pin* é composto por um nome, uma variável booleana que indica se o *pin* deve estar exposto ao exterior do *cluster*, um endereço (que contém um IP e uma porta, com o respectivo protocolo [UDP ou TCP]) e um descritor. (ver figura 4.5).

Foram especificados quatro tipos distintos de descritor (figura 4.6):

- **Vídeo não comprimido**, que é descrito por largura e altura em pixeis, número de fotogramas por segundo, profundidade de cor em bits, tipo de *chroma subsampling* (ver secção 4.4.4.3) e um valor booleano que indica se o vídeo utiliza varredura interlaçada ou progressiva;

- **Áudio não comprimido**, que contém a taxa de amostragem, a profundidade de bits do som e o número de canais;

- ***Datastream***, descritor de uma *stream* de dados genérica para a qual apenas é especificada a largura de banda em bits, utilizado por exemplo para a transmissão de vídeo ou áudio comprimidos;

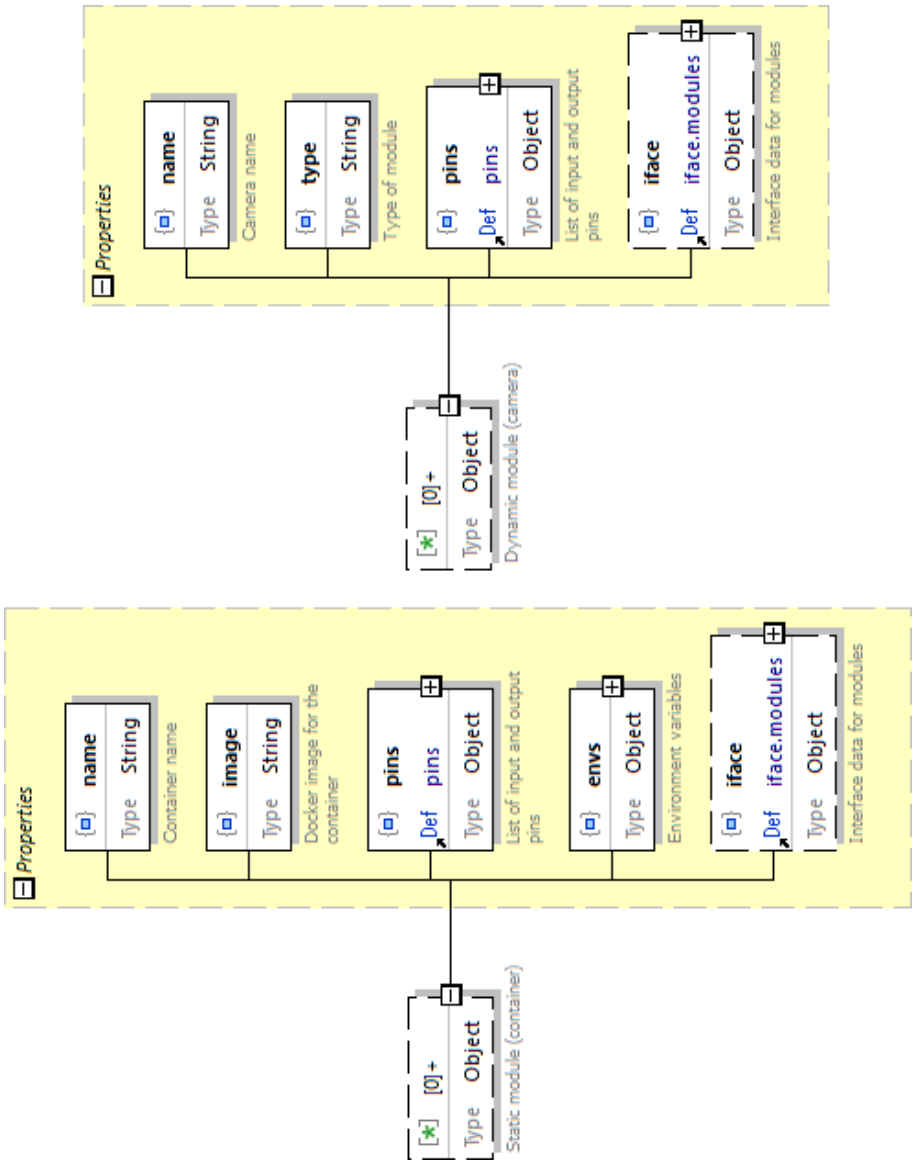

Figura 4.4: Schema - módulos estáticos e dinâmicos

- **Outro**, para tipos de *pins* que não se encaixem em nenhuma das categorias anteriores.

Se por um lado estes descritores são o que torna esta solução mais vocacionada para os estúdios televisivos virtuais, com a definição de novos tipos de descritor esta poderia ser adaptada a outro tipo de aplicação.

Finalmente, para todos os elementos de base (módulos estáticos e dinâmicos, e ligações), existe um objecto que contém os valores para a interface, como a sua posição no *canvas*.

A especificação completa pode ser consultada no anexo B.

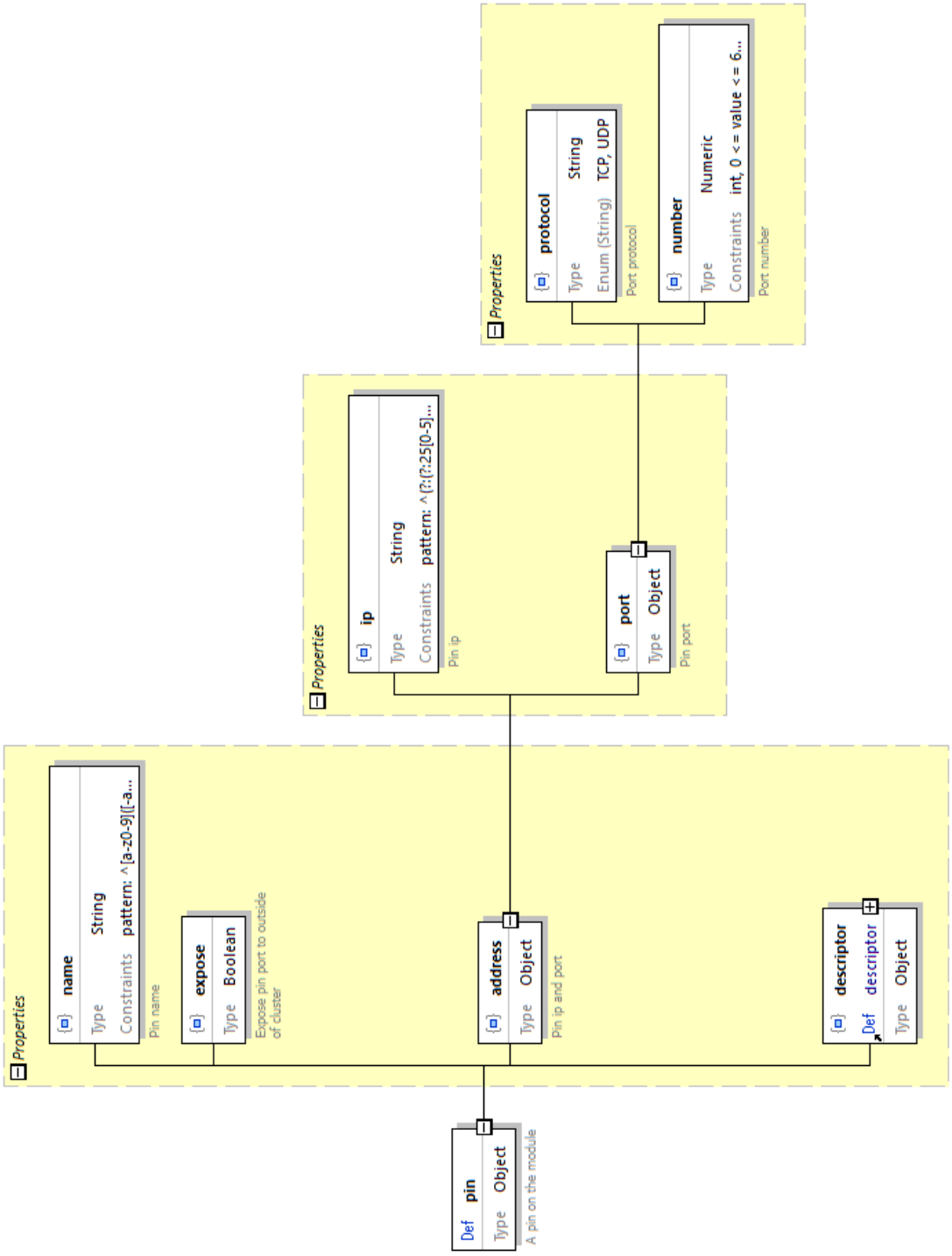

Figura 4.5: Schema - pin

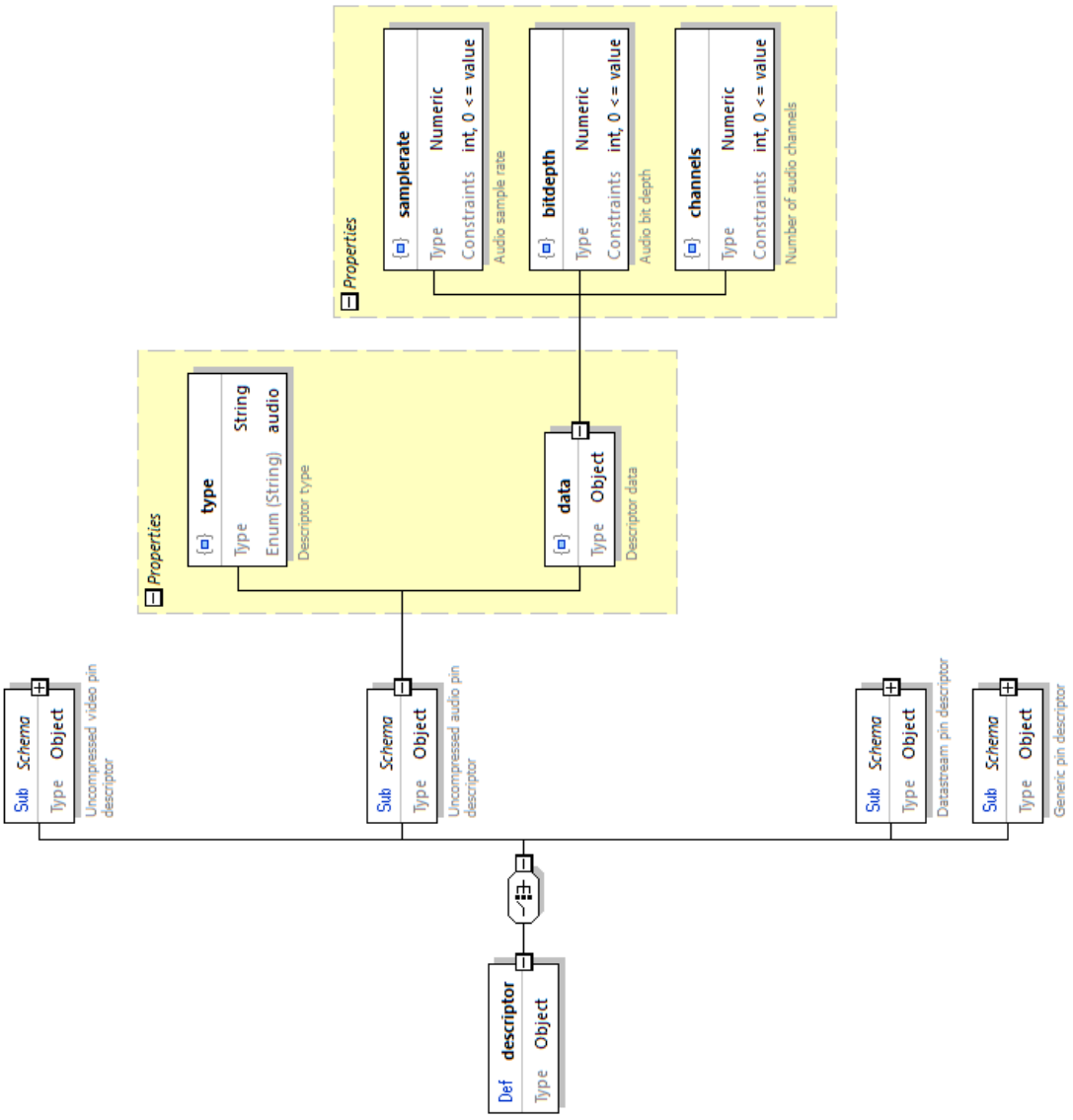

Figura 4.6: Schema - descritor

4.4.3 Interface

Foi desenvolvida em paralelo com este trabalho, pelo colega António Presa [**?**], uma interface Web utilizando a *framework* **React**. Esta permite "desenhar" o sistema desejado, montando e interligando os diferentes módulos da aplicação (ver imagem 4.7).

Esta interface consiste num *canvas* ao centro, onde é possível fazer a montagem e interligação dos módulos. Estes são criados a partir do menu lateral esquerdo, onde estão presentes os módulos estáticos (*containers*) e os dinâmicos, descobertos automaticamente na rede. É também neste menu, na secção *Operations*, que se pode fazer a validação, *deployment* e destruição do cenário, para além de importar e exportar um diagrama (imagem 4.8). Selecionando um dos módulos na área de desenho, abre-se o menu lateral direito, onde é possível editar a configuração de cada módulo. Neste menu podemos ainda encontrar a opção para visualizar os dados de monitorização.

4.4.4 Serviços

Dado que não é possível instanciar diretamente os cenários especificados através da interface, é necessário que esta comunique com um middleware que interprete esta informação, prepare o *deployment*, calculando valores em falta, mapeie os módulos em especificações de *containers* válidas em Kubernetes, e os lance na *cloud*.

Ao conjunto destes serviços chamamos **panamax-deployer**, e a sua função é então servir de ponte de ligação entre a interface e o *cluster*. Está configurado como um *container* de Docker e instanciado no Kubernetes, com várias réplicas para tentar garantir a sua disponibilidade.

A instanciação na *cloud* é feita recorrendo a dois tipos de objetos do Kubernetes (ver secção 2.4.1):

- **Deployments**, responsáveis pela instanciação dos *containers* e manutenção do estado de replicação desejado, detetando falhas na sua execução.
- **Services**, para permitir a exposição das funcionalidades dos módulos para o exterior da *cloud* privada, nomeadamente para a interface Web, e a distribuição da carga entre as réplicas dos mesmos módulos.

Este middleware segue um modelo de micro-serviços em camadas (ver figura 4.9) e foi implementado na linguagem **Python 3**.

4.4.4.1 REST API

Na camada mais acima existe uma API REST que expõe três rotas, com as seguintes funções:

- **/api/validate**, método **POST**: validação do cenário especificado
- **/api/deploy**, método **POST**: instanciação dos módulos
- **/api/deploy**, método **DELETE**: destruição dos módulos

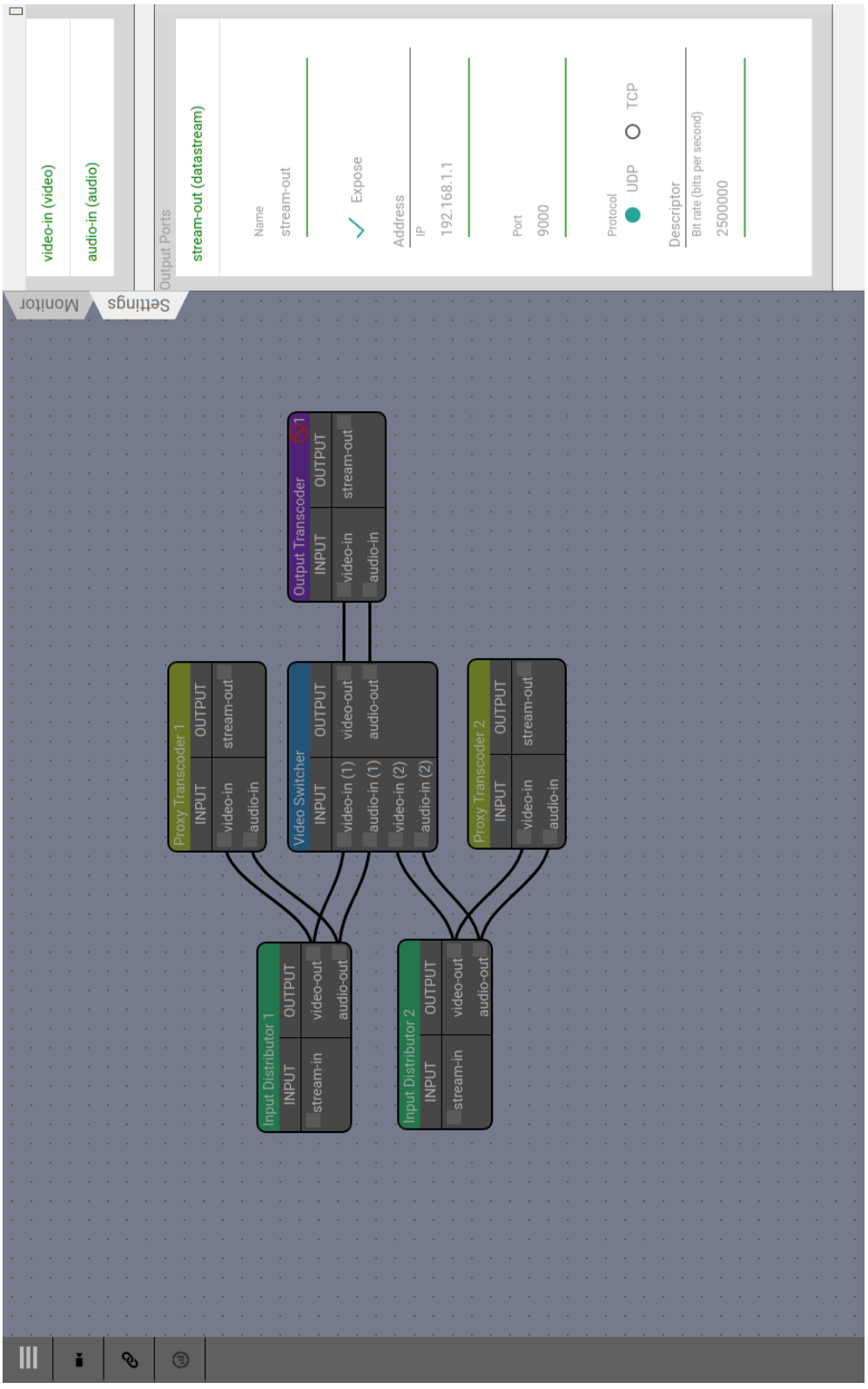

Figura 4.7: Interface do Panamax

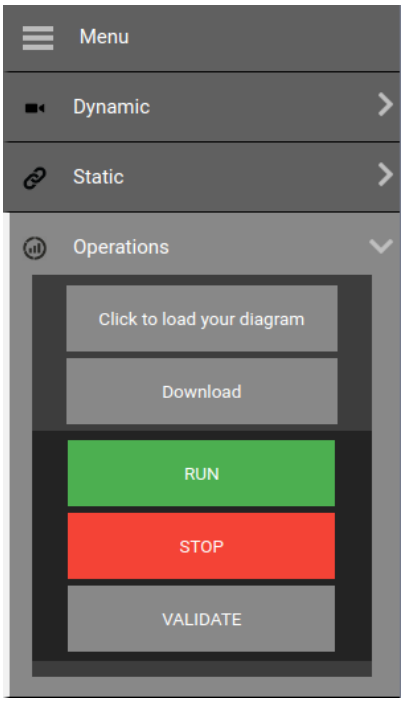

Figura 4.8: Interface do Panamax - *Operations*

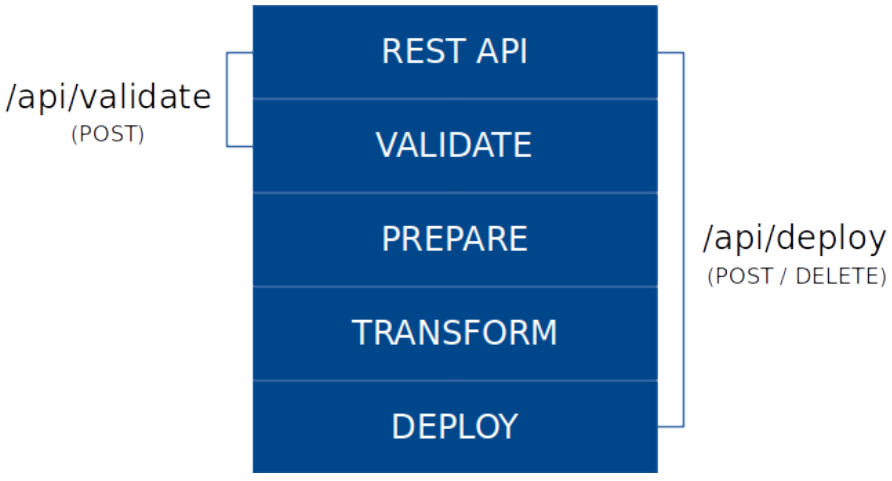

Figura 4.9: Modelo em camadas de serviços do Panamax

Tabela 4.5: API REST - rotas

Endpoint	Método	Corpo do pedido	Corpo da resposta (exemplo)
/api/validate	POST	String JSON	`{"validation": {"success": true}}`
/api/deploy	POST	String JSON	`{ "validation": {"success": true}, "containers": [ "input1": { "success": true, "data":{...} }, ... ] }`
/api/deploy	DELETE	String JSON	`{ "validation": {"success": true}, "containers": [ "input1": { "success": true, "data":{...} }, ... ] }`

Esta foi implementada utilizando a biblioteca **Flask** que permite, de forma sucinta, definir e expor rotas para responder a pedidos REST.

A tabela 4.5 lista as rotas especificadas, bem como os formatos das respostas (versão estendida no anexo D). Nestas vem sempre indicado o resultado da validação do cenário, e, no caso das rotas de *deploy*, é retornada uma lista dos *containers* especificados, com o estado de instanciação / destruição e a informação devolvida pelo orquestrador para cada um deles.

Os diagramas de sequência nas figuras 4.10 e 4.11 detalham o processo de resposta aos pedidos de validação e *deploy*, respectivamente

4.4.4.2 Validate

Nesta camada, em todos os casos, é feita uma validação do cenário comunicado. Recorrendo à biblioteca **jsonschema**, verifica-se se este cumpre todos os parâmetros especificados. Os pedidos de validação terminam neste passo, e os restantes continuam apenas se esta tiver êxito.

4.4.4.3 Prepare

Caso o cenário seja válido segue-se a fase de preparação para *deployment*. É feito o cálculo da largura de banda necessária para cada módulo, dependendo do tipo de descritor.

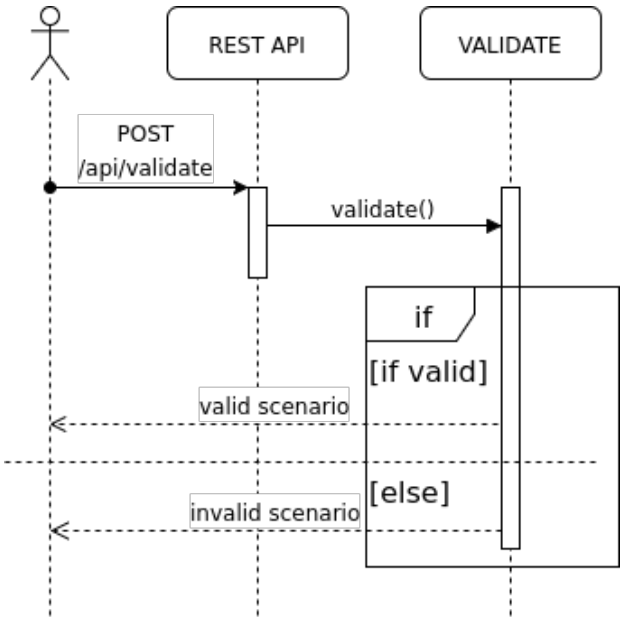

Figura 4.10: Diagrama de sequência - validação

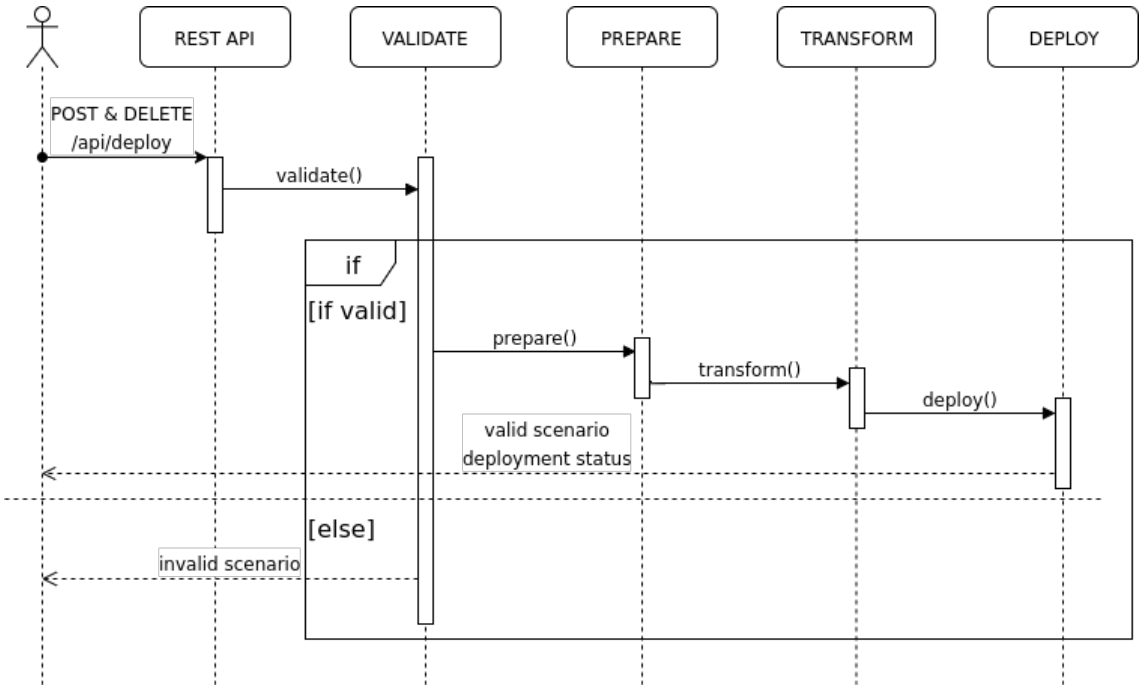

Figura 4.11: Diagrama de sequência - *deploy*

Para vídeo não comprimido, a largura de banda é dada por:

$$w \times h \times f \times c_d \times c_f \tag{4.1}$$

onde w e h representam a largura e altura do vídeo em pixeis, f representa o número de fotogramas por segundo, c_d a profundidade de cor em bits e c_f um fator de ajuste.

Este fator deve-se à aplicação aos vídeos de uma técnica chamada *chroma subsampling*, que aproveita o facto do olho humano não ter tanta capacidade para diferenciar cores como para diferenciar luminosidades. Utiliza-se o formato $Y : C_b : C_r$, em que cada componente tem um valor máximo de quatro, sendo que Y representa a taxa de amostragem de luminância e C_b e C_r a das crominâncias em azul e vermelho, respectivamente [**?**]. Um valor de 4:4:4 representa um fotograma sem *chroma subsampling*, ou seja, todos os valores têm a mesma taxa de amostragem. No formato 4:2:2, os valores de crominância têm metade da amostragem, poupando 33% de largura de banda.

Assim, este fator define-se por:

$$\frac{Y + C_b + C_r}{4} \tag{4.2}$$

Dado que o valor máximo é de 3 vezes a profundidade de cor, multiplicando este fator ao valor em bits da profundidade de cor obtemos a largura de banda total.

Para áudio não comprimido, o cálculo é mais simples:

$$s \times b \times c \tag{4.3}$$

sendo s a taxa de amostragem, b a profundidade de bits do som e c o número de canais de áudio.

Para descritores do tipo *datastream*, a largura de banda consumida é especificada pelo utilizador.

Este valor é reservado na instanciação dos módulos no Kubernetes, através da utilização do sistema de reserva de recursos previamente referido na secção 2.4.1.

4.4.4.4 Transform

A camada de transformação recorre a modelos (*templates*) do Kubernetes. Estes são construídos recorrendo ao sistema de *templates* **Mustache**, e utilizados no serviço através da biblioteca **Pystache** (biblioteca de Mustache para Python).

Foram definidos dois modelos, dado que cada módulo a instanciar necessita, no Kubernetes, de um *service* e de um *deployment*. São percorridos todos os módulos a instanciar, já preparados na camada anterior, e a cada um são aplicados os dois *templates*, resultando assim em dois documentos **YAML** para cada módulo (*service* e *deployment*). Para módulos que não necessitem de expor nenhuma porta, não é instanciado o *service*.

Assim como a definição de diferentes tipos de descritor no *schema* permitiria a instanciação de diferentes tipos de aplicações, a criação de outros *templates* permitiria a utilização de outra solução de orquestração.

4.4.4.5 Deploy

Finalmente, a camada de instanciação no *cluster* utiliza o cliente oficial de Python para o Kubernetes. Esta biblioteca permite aceder à API do orquestrador, definindo classes e métodos para a comunicação com a mesma.

Para cada módulo previamente definido, validado, preparado e transformado, são chamadas as funções de criação / remoção dos *deployments* e *services*.

4.4.5 Monitorização

A monitorização de uma *cloud* não é trivial. Num *cluster* como o proposto, poderá ser desejável recolher e apresentar informação a vários níveis, desde as máquinas físicas aos *containers* que nelas estão instanciadas. Tendo em conta a potencial dimensão de um *cluster*, é necessário não só recolher a informação dos vários pontos como armazená-la e disponibilizá-la de forma agregada e facilmente acessível.

Assim, foi instanciada uma solução para este efeito, descrita na documentação do Kubernetes [**?**], que utiliza os seguintes componentes:

- **cAdvisor**:

  Integrado no **kubelet** (já referido em 2.4.1), este componente encontra-se dessa forma instanciado em todos os nós do *cluster*. É responsável por recolher dados como utilização de CPU, memória e rede de todos os *containers* e da própria máquina, sendo que é o **kubelet** que expõe estes dados através de uma API REST.

- **Heapster**:

  É instanciado no *cluster* como qualquer outra aplicação (numa *Pod*), e tem o papel de recolher os dados de monitorização disponibilizados por cada nó, agrupá-los e enviá-los para um módulo de armazenamento.

- **InfluxDB**:

  Componente de armazenamento (*storage backend*) utilizado nesta solução. É uma base de dados *time series*, indicada para armazenamento de informação que segue uma linha temporal, como é o caso dos dados de monitorização e eventos. Está instanciado no *cluster* como uma *Pod* e providencia uma API para escrita e consulta desta informação.

- **Grafana**:

  Interface de visualização dos dados de monitorização. Recebe os dados do **InfluxDB**, apresentando-os em *dashboards* configuráveis e intuitivos (ver imagem 4.12).

Figura 4.12: Grafana

A figura 4.13 apresenta uma visão geral da arquitetura que, conforme previamente referido, é proposta pelo Kubernetes para fornecer uma solução a este problema, sendo que não é apresentada a componente de visualização da informação.

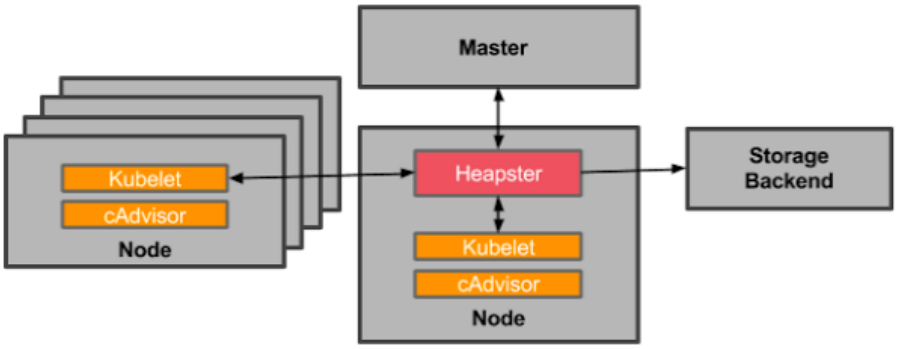

Figura 4.13: Arquitetura da solução de monitorização [**?**]

4.5 Resumo

Neste capítulo foi feita uma descrição do problema inicialmente apresentado, tendo sido explicada a arquitetura de alto nível e os seus requisitos.

Foi feito um resumo das tecnologias utilizadas no desenvolvimento da solução proposta. De seguida, esta foi descrita, tendo sido explicado o processo seguido, começando pela configuração da *cloud*, nomeadamente a instalação e configuração do sistema operativo, instanciação do orquestrador e alguns testes de performance comparativos entre a rede nativa e a *Container Network*. Quanto à implementação da aplicação Panamax, foram listados alguns requisitos adicionais que surgiram durante o trabalho.

Foram abordados os dois tipos de módulos que compõem os cenários bem como o formato criado para os descrever, com uma explicação das definições mais importantes. É feita uma breve abordagem à interface que foi desenvolvida em paralelo com esta dissertação.

De seguida é revisto em maior detalhe o funcionamento das diferentes camadas dos serviços do Panamax. No caso da API REST são apresentados exemplos das respostas devolvidas pela aplicação aos pedidos que são feitos às suas três rotas, bem como diagramas de sequência que ilustram o processamento dos pedidos.

Finalmente é abordada a solução encontrada para a recolha, armazenamento e apresentação dos dados de monitorização da *cloud*, com uma descrição dos seus componentes individuais e visão geral da sua arquitetura.

Capítulo 5

Validação

Neste capítulo pretende-se demonstrar o correto funcionamento da aplicação desenvolvida. Para isto foram lançados dois cenários de estúdios televisivos virtuais, com diferentes configurações.

5.1 Ambiente de teste e Metodologia

Os testes foram realizados na *cloud* privada descrita no capítulo 4. A solução desenvolvida foi também instanciada nesta *cloud*, com duas réplicas a ser executadas.

Procuramos demonstrar o funcionamento do mecanismo de reserva de recursos. Para este efeito, o parâmetro mais relevante é a largura de banda disponível, que influencia em que nós os *containers* podem ser lançados de acordo com a configuração dos descritores dos módulos. Foi configurado em cada nó um recurso opaco correspondente à largura de banda disponível, de acordo com a interface de rede presente na máquina (ver secção 2.4.1 sobre reserva de recursos no Kubernetes).

Foram ainda registados os tempos decorridos em cada etapa nos pedidos à rota */api/deploy*, no método POST, na seguinte sequência:

- Validação
- Preparação
- Transformação e instanciação de cada módulo

Os cenários instanciados assentam todos na mesma arquitetura, definida no capítulo 3. São compostos por dois módulos Input Distributor, três Proxy Transcoders, um Video Switcher e um Output. A diferença entre eles são os descritores utilizados, cujos valores dos parâmetros podem ser consultados na tabela 5.1. Nestes cenários todos os descritores do mesmo tipo têm parâmetros iguais, dado que ambas as *streams*, tanto de entrada como de saída, têm o mesmo formato, e

Tabela 5.1: Cenários testados - descritores

Tipo de descritor	**Pins**	**Parâmetro**	**Cenário um (1080p)**	**Cenário dois (small)**
Datastream	stream-in stream-out	bitrate	13642k	380k
Vídeo não comprimido	video-in video-out	width	1920	176
		height	1080	100
		framerate	50.0	25.0
		colordepth	8	8
		colorspace	4:2:0	4:2:0
		interlaced	false	false

o vídeo não comprimido não sofre alterações às suas características. As *streams* de saída das versões *proxy*, à saída dos *Proxy Transcoders*, não foram tidas em conta.

Ambos os cenários representam a emissão de um evento ao vivo com duas câmaras de vídeo. No cenário um a resolução do vídeo é 1080p (1920 por 1080 pixeis) a 50 FPS. No cenário dois utiliza-se uma resolução baixa, de 176 por 100 pixeis a 25 FPS.

A largura de banda necessária, calculada pela aplicação para cada tipo de módulo, pode ser consultada na tabela 5.2.

Tabela 5.2: Cenários testados - largura de banda

Módulo	**Pins**	**Largura de banda necessária**	
		Cenário um	**Cenário dois**
Input Distributor	stream-in	13.64 Mb/s	0.38 Mb/s
	video-out	1244.16 Mb/s	5.28 Mb/s
	Total	1257.80 Mb/s	5.66 Mb/s
Proxy Transcoder	video-in	1244.16 Mb/s	5.28 Mb/s
	Total		
Video Switcher	video-in1	1244.16 Mb/s	5.28 Mb/s
	video-in2		
	video-out		
	Total	3732.48 Mb/s	15.84 Mb/s
Output	video-in	1244.16 Mb/s	5.28 Mb/s
	stream-out	13.64 Mb/s	0.38 Mb/s
	Total	1257.80 Mb/s	5.66 Mb/s

5.2 Resultados

Na figura 5.1 apresentam-se os resultados do tempo decorrido em cada fase do processo de instanciação do segundo cenário. A duração total da operação foi de cerca de 234ms.

Na etapa de validação foram dispendidos cerca de 5.8ms. O tempo da etapa de preparação é desprezável comparado com as restantes, com 0.1ms. Como se pode verificar, os tempos de transformação dos vários módulos foram sensivelmente constantes, dada a semelhança das operações. Esta fase demorou em média 7.3ms, com um desvio padrão de 0.8ms.

Já o tempo da instanciação varia consideravelmente, potencialmente devido a dois fatores: por um lado é feita uma comunicação à API do Kubernetes, podendo haver atrasos na rede; por outro, para os módulos Video Switcher e Output não é criado um *Service* no Kubernetes, dado que estes não expõem portas para o exterior (conforme referido na secção 4.4.4.4). A média do tempo decorrido nesta etapa foi de 24ms, com um desvio padrão de 11ms. Contudo, em módulos em que é criado um Service decorreram em média 29.9ms, sendo que nos restantes decorreram apenas 9.3ms. Agrupando os resultados desta forma reduz-se o desvio padrão para 5.5ms e 2.4ms, respetivamente.

Os resultados deste teste tendo em conta o cenário um seguem uma distribuição semelhante. Tabelas com os valores detalhados destas medições podem ser consultadas no anexo F.

5.2.1 Cenário um

No cenário um, todos os módulos a instanciar têm um requisito de largura de banda superior a 1Gb/s. Isto significa que apenas deveriam ser instanciados nos nós com placas de rede de 10Gb/s (*CORONA* e *DESPERADOS*).

O seguinte excerto obtido com o cliente de linha de comandos do Kubernetes demonstra este resultado:

```
1  NAME                                  STATUS     NODE
2  input1-3152232280-zgj7h               Running    desperados.mog.local
3  input2-3723247452-wbh61               Running    corona.mog.local
4  output-2189837926-96hhs               Running    desperados.mog.local
5  panamax-deployer-4104198252-6tw8d     Running    corona.mog.local
6  panamax-deployer-4104198252-bzh7r     Running    sagres.mog.local
7  proxy1-2713323557-gvfnc               Running    desperados.mog.local
8  proxy2-3253405737-qtrw8               Running    corona.mog.local
9  proxyout-4074310261-x4dc2             Running    corona.mog.local
10 videoswitcher-1742784718-x9s8h        Running    desperados.mog.local
```

Podemos confirmar que apenas os nós anteriormente mencionados foram utilizados para instanciar *containers* do cenário de teste, não ultrapassando os limites impostos. Os *containers* da aplicação **panamax-deployer** podem estar em qualquer nó, visto não terem requisitos definidos.

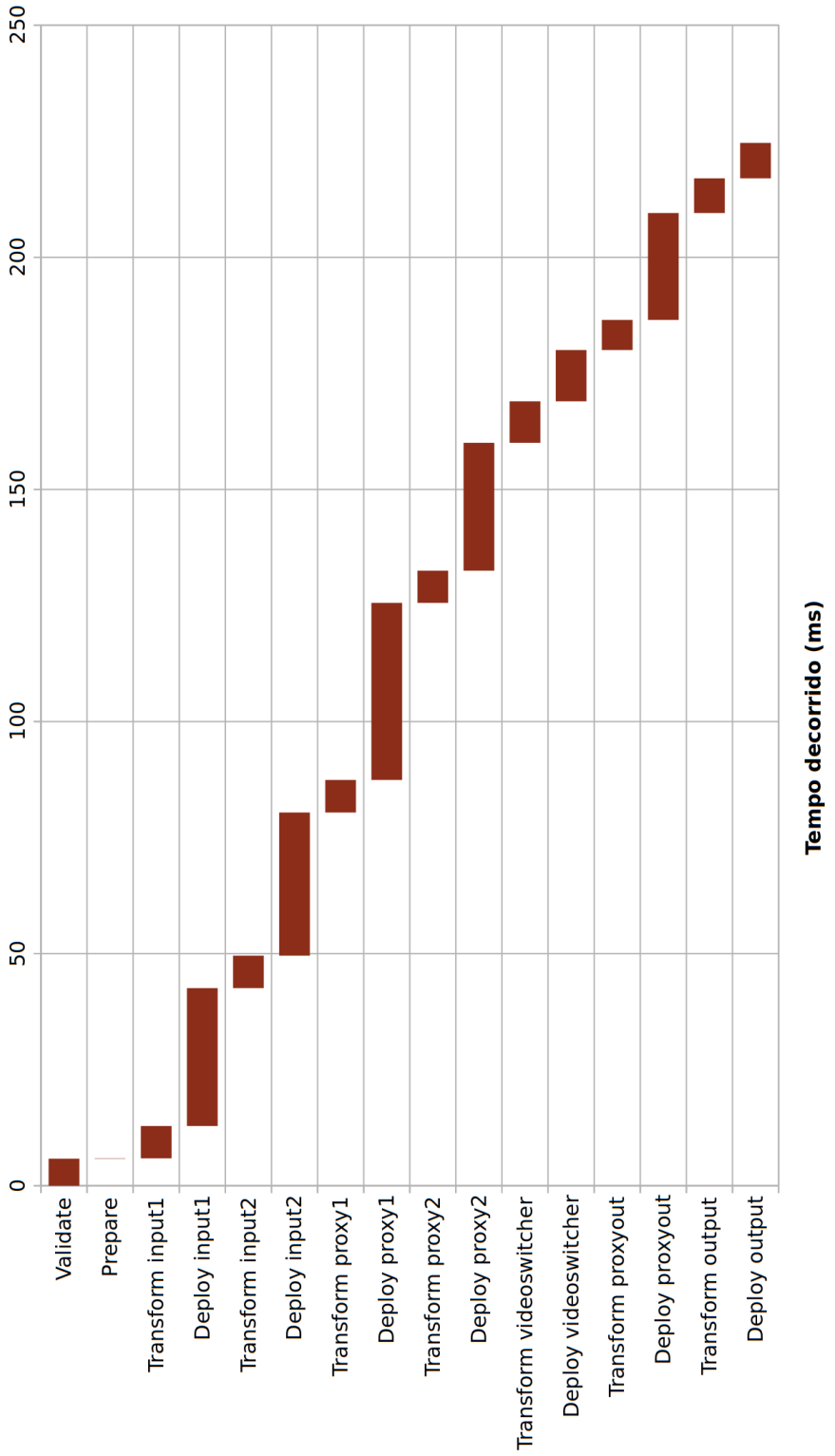

Figura 5.1: Testes - tempo decorrido em cada etapa (ms, cenário dois)

5.2.2 Cenário dois

Já no cenário dois, os baixos requisitos de largura de banda permitem que os módulos sejam instanciados em qualquer um dos nós:

```
1  NAME                                  STATUS     NODE
2  input1-2182364610-rt82p               Running    sagres.mog.local
3  input2-2751282630-3jx4c               Running    desperados.mog.local
4  output-751650000-t14fk                Running    sagres.mog.local
5  panamax-deployer-4104198252-6tw8d     Running    corona.mog.local
6  panamax-deployer-4104198252-bzh7r     Running    sagres.mog.local
7  proxy1-960562841-ppjsg                Running    corona.mog.local
8  proxy2-1498547869-0p8gr               Running    sagres.mog.local
9  proxyout-2011171049-315lg             Running    corona.mog.local
10 videoswitcher-3575564182-63jqf        Running    desperados.mog.local
```

Neste caso podemos verificar que o orquestrador acabou por utilizar também as máquinas com interfaces de 10Gb/s para alguns dos *containers*. Este comportamento pode-se dever à maior capacidade destes nós a nível de outros recursos como CPU e memória.

5.3 Conclusões

Neste capítulo foram configurados dois cenários de teste a partir da interface. Estes foram validados e instanciados pela aplicação, passando por todas as camadas da mesma.

Foram efetuados testes nos quais foi validada a funcionalidade de reserva de recursos, mais propriamente da largura de banda. O valor a reservar é calculado pela aplicação e reservado no orquestrador para cada módulo, a partir do seu descritor. Com um dos cenários a necessitar de recursos apenas disponíveis num pequeno subconjunto do *cluster*, foi possível verificar que apenas as máquinas com suficiente largura de banda seriam utilizadas para a instanciação. A solução poderia ser facilmente adaptada a aplicações com necessidade de diferentes recursos, sendo apenas necessário definir outros tipos de descritores.

Finalmente foi realizado um estudo do tempo dispendido em cada uma das etapas deste processo. Foi possível concluir que o maior gasto de tempo advém da fase de instanciação, dada a necessidade de comunicação com a API do orquestrador. Devido à forma como esta camada está implementada, nos módulos sem a necessidade de exposição de portas não é criado um dos dois objetos do Kubernetes, o Service (como é o caso do Video Switcher e do Output). É assim notório um menor tempo dispendido nessa etapa.

Concluímos que no caso mais simples possível, em que apenas seja instanciado um módulo, é dispendido cerca de 13.7% do tempo na fase de validação, 0.2% na de preparação, 16.3% na de transformação e 69.8% na de instanciação (Service e Deployment), para um tempo total de

aproximadamente 42.6ms. Aumentando o número de módulos, o tempo dispendido nas duas primeiras etapas manter-se-à aproximadamente constante. Será contudo reduzida a sua percentagem relativamente ao tempo total, dada a existência de mais etapas neste processo.

Capítulo 6

Conclusões e Trabalho Futuro

6.1 Conclusões

As tecnologias de virtualização e de computação na *cloud* atualmente disponíveis permitem a rápida implementação e o fácil escalamento de aplicações e serviços, possibilitando um menor investimento inicial em infraestrutura.

A evolução em tecnologias como redes *ethernet* de 10 ou mais gigabit por segundo potenciam a utilização de soluções deste tipo para cenários de produção de conteúdo televisivo, com todos os benefícios da *cloud* e ainda com a particularidade de não serem necessários todos os dispendiosos equipamentos normalmente associados a esta atividade.

Apesar da instanciação de aplicações na *cloud* poder não ser trivial para muitos utilizadores, as plataformas de orquestração oferecem métodos de expansão às suas funcionalidades, que permitem a criação de aplicações auxiliares e *middlewares*.

Nesta dissertação foram analisadas as tecnologias mais apropriadas à criação e configuração de uma *cloud* privada para o *deployment* de aplicações em *containers*.

Foi proposta uma aplicação que, associada a um orquestrador e em paralelo com a interface desenvolvida por António Presa [**?**], permite simplificar o processo de instanciação, orquestração, aprovisionamento e monitorização de aplicações na *cloud*, não só no caso dos estúdios televisivos virtuais como de outros cenários.

Esta aplicação, que corre num *container*, consiste num conjunto de serviços divididos em camadas que expõem os recursos de uma *cloud* para permitir o *deployment* de aplicações *containerizadas*. Os utilizadores operam uma interface Web, desenhando e configurando a aplicação a instanciar. Esta gera um ficheiro de descrição do cenário, que é recebido pela aplicação na camada da API REST. É feita a validação, preparação e transformação do cenário, que é de seguida instanciado em *containers* na *cloud*. Os resultados destas operações, bem como métricas de utilização da *cloud*, são disponibilizados à interface.

Foram efetuados testes à solução apresentada, tendo sido validada a funcionalidade de reserva de recursos. Os requisitos são calculados pela aplicação, de acordo com os descritores especificados para cada *pin*, sendo de seguida reservados no orquestrador. Foi também analisado o tempo

dispendido em cada uma das etapas de instanciação. São apresentadas, nas conclusões do capítulo 3, as percentagens do tempo dispendido em cada etapa relativamente ao tempo total.

Assim, este trabalho contribui para a proliferação das tecnologias de *cloud computing*, tornando-as mais acessíveis ao utilizador final que possa não ter um conhecimento aprofundado de aplicações e sistemas distribuídos.

6.2 Satisfação dos Objetivos

Os objetivos inicialmente propostos foram totalmente satisfeitos. A solução implementada permitiu validar a aplicabilidade da arquitetura definida ao caso de teste dos estúdios televisivos virtuais.

Apesar de ter sido escolhido um conjunto de soluções que se adequavam ao caso estudado e de estas satisfazerem os requisitos deste, seria possível a utilização da arquitetura definida com outras soluções de orquestração e para outros cenários de teste, conforme referido na secção 4.4.4.

6.3 Trabalho Futuro

Tendo sido proposta uma arquitetura de alto nível, adaptável a diferentes cenários e ferramentas, esta solução pode dar início a uma série de trabalhos, estudos e investigações que podem trazer contributo científico na área da computação na *cloud*.

Quanto à generalização da solução encontrada a outros tipos de aplicação, podem ser implementados mais tipos de descritor, que exponham de forma detalhada os requisitos de funcionamento de cada componente de um sistema.

Também neste aspeto a utilização de trabalho relacionado na área das *Architecture description language* (ADL) poderá ser interessante não só na definição mais formal da arquitetura desta plataforma [**?**] como na sua monitorização, diagnóstico e deteção de problemas [**?**, **?**, **?**].

Ao nível do orquestrador, mais das suas funcionalidades podem ser integradas na solução, como a utilização de volumes de armazenamento permanentes, o que permitiria instanciar outros tipos de aplicação.

As aplicações instanciadas através desta solução calculam a largura de banda necessária a partir do descritor. Seria interessante investigar como calcular consumos de outros recursos, como CPU ou memória RAM, permitindo garantir a correta alocação destes aos *containers*.

Na solução proposta não foram tidos em conta problemas como segurança e encriptação dos canais de comunicação. Durante o desenvolvimento desta dissertação foram disponibilizadas no Kubernetes (na versão 1.6) funcionalidades de controlo de acesso baseado em perfis (*Role-based access control*). A utilização destas tecnologias poderia ser importante num cenário de produção real e seria interessante investigar melhor as potencialidades das ferramentas utilizadas a este nível.

Com a recente entrada da Microsoft no mercado dos *containers*, seria interessante investigar as potencialidades do Windows para este e outros casos de uso, que tenham eventualmente sido desenvolvidos para este sistema operativo.

Anexo A

Especificações das máquinas

Tabela A.1: Especificações técnicas das máquinas utilizadas

	MAKE/MODEL	**CPU**	**RAM**	**GPU**	**NETWORK**
CORONA	MOG Xpress	Intel Core i7-4790S	16GB	Intel iGPU	Intel X550 10Gb
DESPERADOS	MOG Xpress	Intel Core i7-4790S	16GB	Intel iGPU	Intel X550 10Gb
CRISTAL	MOG Xpress	Intel Core i7-2600	4GB	Intel iGPU	Intel 1Gb
IMPERIAL	MOG Xpress	Intel Core i7-2600K	4GB	Intel iGPU	Intel 1Gb
DUVEL	Dell Poweredge R310	Intel Xeon X3470	8GB	Matrox G200eW	Qlogic 1GB
SAGRES	Dell Poweredge R410	2x Intel Xeon E5640	32GB	Matrox G200eW	Broadcom 1Gb
SUPERBOCK	Dell Poweredge R430	2x Intel Xeon E5-2620 v3	32GB	Matrox G200eR	Broadcom 1Gb

Anexo B

JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Application scenario",
  "type": "object",
  "properties": {
    "containers": {
      "$ref": "#/definitions/containers"
    },
    "cameras": {
      "$ref": "#/definitions/cameras"
    },
    "connections": {
      "$ref": "#/definitions/connections"
    }
  },
  "required": [
    "containers",
    "cameras",
    "connections"
  ],
  "additionalProperties": false,
  "definitions": {
    "pin": {
      "title": "Pin",
      "description": "A pin on the module",
      "type": "object",
      "properties": {
        "name": {
          "description": "Pin name",
          "type": "string",
          "pattern": "^[a-z0-9]([-a-z0-9]*[a-z0-9])?$"
        },
        "expose": {
```

```
34          "description": "Expose pin port to outside of cluster",
35          "type": "boolean"
36        },
37        "address": {
38          "description": "Pin ip and port",
39          "type": "object",
40          "properties": {
41            "ip": {
42              "description": "Pin ip",
43              "type": "string",
44              "pattern": "^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.)
                    {3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$"
45            },
46            "port": {
47              "description": "Pin port",
48              "type": "object",
49              "properties": {
50                "protocol": {
51                  "description": "Port protocol",
52                  "type": "string",
53                  "enum": [
54                    "TCP",
55                    "UDP"
56                  ]
57                },
58                "number": {
59                  "description": "Port number",
60                  "type": "integer",
61                  "minimum": 0,
62                  "maximum": 65535
63                }
64              },
65              "required": [
66                "protocol",
67                "number"
68              ],
69              "additionalProperties": false
70            }
71          },
72          "required": [
73            "ip",
74            "port"
75          ],
76          "additionalProperties": false
77        },
78        "descriptor": {
79          "$ref": "#/definitions/descriptor"
80        }
81      },
```

```
82      "required": [
83        "name",
84        "expose",
85        "address",
86        "descriptor"
87      ],
88      "additionalProperties": false
89    },
90    "descriptor": {
91      "type": "object",
92      "additionalProperties": false,
93      "oneOf": [
94        {
95          "title": "Descriptor",
96          "description": "Uncompressed video pin descriptor",
97          "type": "object",
98          "properties": {
99            "type": {
100             "description": "Descriptor type",
101             "type": "string",
102             "enum": [
103               "video"
104             ]
105           },
106           "data": {
107             "description": "Descriptor data",
108             "type": "object",
109             "properties": {
110               "width": {
111                 "description": "Video width",
112                 "type": "integer",
113                 "minimum": 0
114               },
115               "height": {
116                 "description": "Video height",
117                 "type": "integer",
118                 "minimum": 0
119               },
120               "framerate": {
121                 "description": "Video framerate",
122                 "type": "number",
123                 "minimum": 0
124               },
125               "colordepth": {
126                 "description": "Video colordepth",
127                 "type": "integer",
128                 "minimum": 0
129               },
130               "colorspace": {
```

```
                "description": "Video colorspace",
                "type": "string",
                "enum": [
                  "4:4:4",
                  "4:2:2",
                  "4:2:1",
                  "4:1:1",
                  "4:2:0",
                  "4:1:0",
                  "3:1:1"
                ]
              },
              "interlaced": {
                "description": "Video is interlaced",
                "type": "boolean"
              }
            },
            "required": [
              "width",
              "height",
              "framerate",
              "colordepth",
              "colorspace",
              "interlaced"
            ],
            "additionalProperties": false
          }
        },
        "required": [
          "type",
          "data"
        ],
        "additionalProperties": false
      },
      {
        "title": "Descriptor",
        "description": "Uncompressed audio pin descriptor",
        "type": "object",
        "properties": {
          "type": {
            "description": "Descriptor type",
            "type": "string",
            "enum": [
              "audio"
            ]
          },
          "data": {
            "description": "Descriptor data",
            "type": "object",
```

```
            "properties": {
              "samplerate": {
                "description": "Audio sample rate",
                "type": "integer",
                "minimum": 0
              },
              "bitdepth": {
                "description": "Audio bit depth",
                "type": "integer",
                "minimum": 0
              },
              "channels": {
                "description": "Number of audio channels",
                "type": "integer",
                "minimum": 0
              }
            },
            "required": [
              "samplerate",
              "bitdepth",
              "channels"
            ],
            "additionalProperties": false
          }
        },
        "required": [
          "type",
          "data"
        ],
        "additionalProperties": false
      },
      {
        "title": "Descriptor",
        "description": "Datastream pin descriptor",
        "type": "object",
        "properties": {
          "type": {
            "description": "Descriptor type",
            "type": "string",
            "enum": [
              "datastream"
            ]
          },
          "data": {
            "description": "Descriptor data",
            "type": "object",
            "properties": {
              "bitrate": {
                "description": "Datastream bitrate",
```

```
                "type": "integer",
                "minimum": 0
              }
            },
            "required": [
              "bitrate"
            ],
            "additionalProperties": false
          }
        },
        "required": [
          "type",
          "data"
        ],
        "additionalProperties": false
      },
      {
        "title": "Descriptor",
        "description": "Generic pin descriptor",
        "type": "object",
        "properties": {
          "type": {
            "description": "Descriptor type",
            "type": "string",
            "enum": [
              "other"
            ]
          },
          "data": {
            "description": "Descriptor data",
            "type": "object"
          }
        },
        "required": [
          "type",
          "data"
        ],
        "additionalProperties": false
      }
    ]
  },
  "pins": {
    "title": "Pins",
    "description": "List of input and output pins",
    "type": "object",
    "properties": {
      "in": {
        "description": "Input pins",
        "type": "array",
```

```
          "items": {
            "$ref": "#/definitions/pin"
          }
        },
        "out": {
          "description": "Output pins",
          "type": "array",
          "items": {
            "$ref": "#/definitions/pin"
          }
        }
      },
      "required": [
        "in",
        "out"
      ],
      "additionalProperties": false
    },
    "containers": {
      "title": "Containers",
      "description": "List of containers (static modules)",
      "type": "array",
      "items": {
        "description": "Static module (container)",
        "type": "object",
        "properties": {
          "name": {
            "description": "Container name",
            "type": "string"
          },
          "image": {
            "description": "Docker image for the container",
            "type": "string"
          },
          "pins": {
            "$ref": "#/definitions/pins"
          },
          "envs": {
            "description": "Environment variables",
            "type": "object"
          },
          "iface": {
            "$ref": "#/definitions/iface.modules"
          }
        },
        "required": [
          "name",
          "image",
          "pins",
```

```
          "envs"
        ],
        "additionalProperties": false
      }
    },
    "cameras": {
      "title": "Cameras",
      "description": "List of cameras (dynamic modules)",
      "type": "array",
      "items": {
        "description": "Dynamic module (camera)",
        "type": "object",
        "properties": {
          "name": {
            "description": "Camera name",
            "type": "string"
          },
          "type": {
            "description": "Type of module",
            "type": "string"
          },
          "pins": {
            "$ref": "#/definitions/pins"
          },
          "iface": {
            "$ref": "#/definitions/iface.modules"
          }
        },
        "required": [
          "name",
          "type",
          "pins"
        ],
        "additionalProperties": false
      },
      "additionalProperties": false
    },
    "endpoint": {
      "title": "Endpoint",
      "description": "A pin and its module",
      "type": "object",
      "properties": {
        "module": {
          "description": "Endpoint module",
          "type": "string"
        },
        "pin": {
          "description": "Endpoint pin",
          "type": "string"
```

```
        }
      },
      "required": [
        "module",
        "pin"
      ],
      "additionalProperties": false
    },
    "connection": {
      "title": "Connection",
      "description": "A connection between two endpoints",
      "type": "object",
      "properties": {
        "source": {
          "$ref": "#/definitions/endpoint"
        },
        "destination": {
          "$ref": "#/definitions/endpoint"
        },
        "iface": {
          "$ref": "#/definitions/iface.connections"
        }
      },
      "required": [
        "source",
        "destination"
      ],
      "additionalProperties": false
    },
    "connections": {
      "title": "Connections",
      "description": "List of connections",
      "type": "array",
      "items": {
        "$ref": "#/definitions/connection"
      }
    },
    "iface.position": {
      "title": "Position",
      "description": "A position on the interface",
      "type": "object",
      "properties": {
        "x": {
          "description": "x value of position",
          "type": "number"
        },
        "y": {
          "description": "y value of position",
          "type": "number"
```

```
        }
      },
      "required": [
        "x",
        "y"
      ],
      "additionalProperties": false
    },
    "iface.modules": {
      "title": "Modules Interface",
      "description": "Interface data for modules",
      "type": "object",
      "properties": {
        "position": {
          "$ref": "#/definitions/iface.position"
        }
      },
      "required": [
        "position"
      ],
      "additionalProperties": false
    },
    "iface.connections": {
      "title": "Connections Interface",
      "description": "Interface data for connections",
      "type": "object",
      "properties": {
        "points": {
          "description": "List of points on connection",
          "type": "array",
          "items": {
            "$ref": "#/definitions/iface.position"
          }
        }
      },
      "required": [
        "points"
      ],
      "additionalProperties": false
    }
  }
}
```

Anexo C

Templates de transformação

C.1 Service

```
apiVersion: v1
kind: Service
metadata:
  name: {{name}}
  labels:
    run: {{name}}
spec:
  type: NodePort
  ports:
  {{#pins.in}}
  {{#expose}}
  - name: {{name}}
    port: {{address.port.number}}
    protocol: {{address.port.protocol}}
  {{/expose}}
  {{/pins.in}}
  {{#pins.out}}
  {{#expose}}
  - name: {{name}}
    port: {{address.port.number}}
    protocol: {{address.port.protocol}}
  {{/expose}}
  {{/pins.out}}
  selector:
    run: {{name}}
```

C.2 Deployment

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: {{name}}
spec:
  template:
    metadata:
      labels:
        run: {{name}}
    spec:
      containers:
      - name: {{name}}
        image: {{image}}
        resources:
          requests:
            {{#bandwidth}}
              pod.alpha.kubernetes.io/opaque-int-resource-bandwidth: {{
                  bandwidth}}
            {{/bandwidth}}
        ports:
          {{#pins.in}}
          - name: {{name}}
            containerPort: {{address.port.number}}
            protocol: {{address.port.protocol}}
          {{/pins.in}}
          {{#pins.out}}
          - name: {{name}}
            containerPort: {{address.port.number}}
            protocol: {{address.port.protocol}}
          {{/pins.out}}
        env:
          {{#envs}}
          - name: {{name}}
            value: "{{value}}"
          {{/envs}}
```

Anexo D

Respostas da API

D.1 /api/deploy [POST]


```
{
  "validation": {
    "success": true
  },
  "containers": [
    {
      "success": true,
      "operation": "create",
      "name": "input1",
      "kind": "service",
      "data": {
        "api_version": "v1",
        "kind": "Service",
        "metadata": {
          "annotations": null,
          "cluster_name": null,
          "creation_timestamp": "2017-06-21T13:56:00+00:00",
          "deletion_grace_period_seconds": null,
          "deletion_timestamp": null,
          "finalizers": null,
          "generate_name": null,
          "generation": null,
          "labels": {
            "run": "input1"
          },
          "name": "input1",
          "namespace": "default",
          "owner_references": null,
          "resource_version": "857871",
          "self_link": "/api/v1/namespaces/default/services/input1",
          "uid": "5ba67c8c-5689-11e7-a141-54bef7656421"
```

```
        },
        "spec": {
          "cluster_ip": "10.107.69.92",
          "deprecated_public_i_ps": null,
          "external_i_ps": null,
          "external_name": null,
          "load_balancer_ip": null,
          "load_balancer_source_ranges": null,
          "ports": [
            {
              "name": "stream-in",
              "node_port": 31479,
              "port": 2000,
              "protocol": "UDP",
              "target_port": "2000"
            }
          ],
          "selector": {
            "run": "input1"
          },
          "session_affinity": "None",
          "type": "NodePort"
        },
        "status": {
          "load_balancer": {
            "ingress": null
          }
        }
      }
    },
    {
      "success": false,
      "operation": "create",
      "name": "input1",
      "kind": "deployment",
      "data": {
        "kind": "Status",
        "apiVersion": "v1",
        "metadata": {},
        "status": "Failure",
        "message": "object is being deleted: deployments.extensions \"
            input1\" already exists",
        "reason": "AlreadyExists",
        "details": {
          "name": "input1",
          "group": "extensions",
          "kind": "deployments"
        },
        "code": 409
```

```
      }
    },
...
```

D.2 /api/deploy [DELETE]

```
{
  "containers": [
    {
      "success": true,
      "operation": "delete",
      "name": "input1",
      "kind": "service",
      "data": {
        "api_version": "v1",
        "code": 200,
        "details": null,
        "kind": "Status",
        "message": null,
        "metadata": {
          "resource_version": null,
          "self_link": null
        },
        "reason": null,
        "status": "Success"
      }
    },
    {
      "success": false,
      "operation": "delete",
      "name": "input1",
      "kind": "deployment",
      "data": {
        "kind": "Status",
        "apiVersion": "v1",
        "metadata": {},
        "status": "Failure",
        "message": "deployments.extensions \"input1\" not found",
        "reason": "NotFound",
        "details": {
          "name": "input1",
          "group": "extensions",
          "kind": "deployments"
        },
        "code": 404
      }
```

```
    },
...
```

D.3 /api/validate [POST]

```
{
  "validation": {
    "success": false
  }
}
```

Anexo E

Exemplo de cenário

E.1 Proveniente da interface

```
{
  "containers": [
    {
      "name": "input1",
      "image": "docker.mog.local:5000/inputdistributor:0.3-vgoncalves",
      "pins": {
        "in": [
          {
            "name": "video-in",
            "expose": true,
            "address": {
              "ip": "192.168.1.2",
              "port": {
                "protocol": "UDP",
                "number": 2000
              }
            },
            "descriptor": {
              "type": "video",
              "data": {
                "width": 1920,
                "height": 1080,
                "framerate": 24.0,
                "colordepth": 8,
                "colorspace": "4:2:2",
                "interlaced": false
              }
            }
          },
          {
            "name": "audio-in",
```

```
32            "expose": true,
33            "address": {
34              "ip": "192.168.1.2",
35              "port": {
36                "protocol": "UDP",
37                "number": 2002
38              }
39            },
40            "descriptor": {
41              "type": "audio",
42              "data": {
43                "samplerate": 48000,
44                "bitdepth": 16,
45                "channels": 2
46              }
47            }
48          }
49        ],
50        "out": [
51          {
52            "name": "video-out",
53            "expose": false,
54            "address": {
55              "ip": "192.168.1.145",
56              "port": {
57                "protocol": "UDP",
58                "number": 4000
59              }
60            },
61            "descriptor": {
62              "type": "datastream",
63              "data": {
64                "bitrate": 345678
65              }
66            }
67          },
68          {
69            "name": "http",
70            "expose": true,
71            "address": {
72              "ip": "127.0.0.1",
73              "port": {
74                "protocol": "TCP",
75                "number": 80
76              }
77            },
78            "descriptor": {
79              "type": "other",
80              "data": {}
```

```
            }
          }
        ]
      },
      "envs": {
        "multicastAddrIP": "225.2.2.0",
        "multicastAddrPort": 3000
      }
    }
  ],
  "cameras": [
    {
      "name": "camera1",
      "type": "camera",
      "iface": {
        "position": {
          "x": 4,
          "y": 6
        }
      },
      "pins": {
        "in": [],
        "out": [
          {
            "name": "video-out",
            "expose": false,
            "address": {
              "ip": "192.168.1.145",
              "port": {
                "protocol": "UDP",
                "number": 4000
              }
            },
            "descriptor": {
              "type": "datastream",
              "data": {
                "bitrate": 345678
              }
            }
          }
        ]
      }
    }
  ],
  "connections": [
    {
      "source": {
        "module": "input1",
        "pin": "video-out"
```

```
      },
      "destination": {
        "module": "proxy1",
        "pin": "video-in"
      },
      "iface": {
        "points": [
          {
            "x": 5,
            "y": 6
          }
        ]
      }
    },
    {
      "source": {
        "module": "input2",
        "pin": "video-out"
      },
      "destination": {
        "module": "proxy2",
        "pin": "video-in"
      }
    }
  ]
}
```

E.2 Transformado para instanciação

E.2.1 Service

```
apiVersion: v1
kind: Service
metadata:
  labels: {run: input1}
  name: input1
spec:
  ports:
  - {name: video-in, port: 2000, protocol: UDP}
  - {name: audio-in, port: 2002, protocol: UDP}
  - {name: http, port: 80, protocol: TCP}
  selector: {run: input1}
  type: NodePort
```

E.2.2 Deployment

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata: {name: input1}
spec:
  template:
    metadata:
      labels: {run: input1}
    spec:
      containers:
      - env:
        - {name: multicastAddrIP, value: 225.2.2.0}
        - {name: multicastAddrPort, value: '3000'}
        image: docker.mog.local:5000/inputdistributor:0.3-vgoncalves
        name: input1
        ports:
        - {containerPort: 2000, name: video-in, protocol: UDP}
        - {containerPort: 2002, name: audio-in, protocol: UDP}
        - {containerPort: 4000, name: video-out, protocol: UDP}
        - {containerPort: 80, name: http, protocol: TCP}
        resources:
          requests: {pod.alpha.kubernetes.io/opaque-int-resource-bandwidth:
              798144078}
```

Exemplo de cenário

Anexo F

Resultados dos testes

F.1 Tempo decorrido

F.1.1 Cenário um

Tabela F.1: Testes - tempo decorrido em cada etapa (ms, cenário um)

TASK		START	END	DURATION
Validate		0.000	4.999	4.999
Prepare		4.999	5.063	0.064
Input Distributor (1)	Transform	5.063	11.750	6.687
	Deploy	11.750	32.789	21.040
Input Distributor (2)	Transform	32.789	40.436	7.647
	Deploy	40.436	78.119	37.683
Proxy Transcoder (1)	Transform	78.119	85.047	6.928
	Deploy	85.047	109.974	24.927
Proxy Transcoder (2)	Transform	109.974	117.230	7.256
	Deploy	117.230	139.457	22.227
Video Switcher	Transform	139.457	148.480	9.023
	Deploy	148.480	157.979	9.499
Proxy Transcoder (Output)	Transform	157.979	164.914	6.935
	Deploy	164.914	214.736	49.822
Output	Transform	214.736	221.863	7.127
	Deploy	221.863	233.914	12.051

F.1.2 Cenário dois

Tabela F.2: Testes - tempo decorrido em cada etapa (ms, cenário dois)

TASK		START	END	DURATION
Validate		0.000	5.841	5.841
Prepare		5.841	5.915	0.074
Input Distributor (1)	Transform	5.915	12.879	6.963
	Deploy	12.879	42.583	29.704
Input Distributor (2)	Transform	42.583	49.566	6.983
	Deploy	49.566	80.408	30.842
Proxy Transcoder (1)	Transform	80.408	87.414	7.006
	Deploy	87.414	125.569	38.155
Proxy Transcoder (2)	Transform	125.569	132.505	6.936
	Deploy	132.505	160.049	27.544
Video Switcher	Transform	160.049	168.981	8.932
	Deploy	168.981	180.031	11.050
Proxy Transcoder (Output)	Transform	180.031	186.523	6.492
	Deploy	186.523	209.544	23.021
Output	Transform	209.544	217.024	7.479
	Deploy	217.024	224.650	7.627

Referências

[App16] Appium Team. Appium Introduction. http://appium.io/introduction.html?lang=en, February 2016.

[Car14] Valentín Carela Español. *Network traffic classification : from theory to practice*. PhD thesis, 2014. URL: http://hdl.handle.net/10803/283573.

[DGS11] Maurizio Dusi, Francesco Gringoli e Luca Salgarelli. Quantifying the accuracy of the ground truth associated with Internet traffic traces. *Computer Networks*, 55(5):1158–1167, apr 2011. URL: http://www.sciencedirect.com/science/article/pii/S1389128610003579, doi:10.1016/j.comnet.2010.11.006.

[FT00] Roy T. Fielding e Richard N. Taylor. Principled design of the modern Web architecture. In *Proceedings of the 22nd international conference on Software engineering - ICSE '00*, pages 407–416, New York, New York, USA, jun 2000. ACM Press. URL: http://dl.acm.org/citation.cfm?id=337180.337228, doi:10.1145/337180.337228.

[Gar16] Gartner. Worldwide device shipments. https://www.gartner.com/newsroom/id/2954317, February 2016.

[GNAM13] Lorenzo Gomez, Iulian Neamtiu, Tayyaba Azim e Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 72–81. IEEE, 2013.

[Goo16] Google Inc. Testing Support Library. https://developer.android.com/tools/testing-support-library/index.html, February 2016.

[IAN16a] IANA. Hypertext Transfer Protocol (HTTP) Status Code Registry. https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml, February 2016.

[IAN16b] IANA. Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, February 2016.

[Ind15] Cisco Visual Networking Index. Global traffic forecast 2014-2019. *White Paper, May*, 2015.

[KBFC04] Thomas Karagiannis, Andre Broido, Michalis Faloutsos e Kc Claffy. Transport layer identification of P2P traffic. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement - IMC '04*, page 121, New York, New York, USA, oct 2004. ACM Press. URL: http://dl.acm.org/citation.cfm?id=1028788.1028804, doi:10.1145/1028788.1028804.

[KCF+08] Hyunchul Kim, KC Claffy, Marina Fomenkov, Dhiman Barman, Michalis Faloutsos e KiYoung Lee. Internet traffic classification demystified. In *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08*, pages 1–12, New York, New York, USA, dec 2008. ACM Press. URL: http://dl.acm.org/citation.cfm?id=1544012.1544023, doi:10.1145/1544012.1544023.

[KPF05] Thomas Karagiannis, Konstantina Papagiannaki e Michalis Faloutsos. BLINC. *ACM SIGCOMM Computer Communication Review*, 35(4):229, oct 2005. URL: http://dl.acm.org/citation.cfm?id=1090191.1080119, doi:10.1145/1090191.1080119.

[NA08] Thuy Nguyen e Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4738466, doi:10.1109/SURV.2008.080406.

[Sel16] Selendroid. Selendroid: Selenium for Android. http://selendroid.io/, February 2016.

[Sta16] StatCounter Global Stats. Top 8 Operating Systems from Jan 2010 to Dec 2015. http://gs.statcounter.com/{#}all-os-ww-monthly-201101-201512, February 2016.

[WZA06] Nigel Williams, Sebastian Zander e Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *ACM SIGCOMM Computer Communication Review*, 36(5):5, oct 2006. URL: http://dl.acm.org/citation.cfm?id=1163593.1163596, doi:10.1145/1163593.1163596.

[ZNA05] S. Zander, T. Nguyen e G. Armitage. Automated traffic classification and application identification using machine learning. In *The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)l*, pages 250–257. IEEE, 2005. URL: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1550864, doi:10.1109/LCN.2005.35.